



UNIVERSITÀ DEGLI STUDI DI MILANO

FACOLTÀ DI SCIENZE E TECNOLOGIE

CORSO DI LAUREA TRIENNALE IN
SICUREZZA DEI SISTEMI E DELLE RETI INFORMATICHE

DNS-Based Detection of Network Scanning: a case study analysis

Relatore: Prof. Stelvio Cimato

Tesi di:

Stefano Walter MAVILIO

Matricola: 987993

Anno Accademico 2023 - 2024

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my thesis supervisor, Stelvio Cimato, whose invaluable guidance and dedication have greatly shaped the course of my thesis. I would like to extend my profound gratitude to my family, especially my mom, whose unwavering support has been my strongest foundation throughout this journey. Her belief in me has given me the strength to persevere.

To my closest friends, thank you for your constant encouragement and for always being by my side.

A heartfelt thank you to my girlfriend Giulia for her love and patience, which have been a constant source of motivation.

Lastly, but by no means least, a special thanks to my amazing classmates – Fabrizio, Stefano, Jacopo, Marco, Luca, and Biagio – for making this experience both rewarding and memorable. Thank you all for being a part of this chapter in my life.

Index

Acknowledgements	1
1 Introduction	4
1.1 Contextualization	4
1.2 Thesis Objectives	5
1.3 Relevance of the Research	5
1.4 Structure of the Thesis	5
2 Network Scanning: Concepts and Techniques	7
2.1 Overview: why is Network Security so vital?	7
2.2 Definition and Historical Perspective of Network Scanning	9
2.3 Reconnaissance	12
2.4 Network Scanning Strategies	17
2.5 Host Discovery	22
2.6 Port Scanning	24
2.7 Service, Version and OS Detection	34
3 Detection of Network Scanning	35
3.1 What is Detection?	35
3.2 Internal VS External Detection of Network Attacks	36
3.3 Firewalls	37
3.4 IDS and IPS	42
3.5 Network Traffic Analysis	44
3.6 Classification of Detection Algorithms	46
4 DNS-based Detection of Network Scanning	51
4.1 History of DNS	51
4.2 DNS Basics	51
4.3 DNS Query Resolution	55
4.4 Reverse DNS Query Resolution	57
4.5 DNS Caching	58
4.6 DNS Packet Structure	60
4.7 Proposed DNS-Based Detection Approach	63
5 Case study	69

5.1 Laboratory Setup	69
5.2 Detection Algorithm Implementation	74
5.3 Understanding the Core Logic: The process_packet Function	80
6 Analysis and results	82
6.1 Experimental Execution and Results	82
6.2 Performance Analysis of the Scan Detection System	91
6.3 Limitations and Future Improvements	92
7 Conclusion	94
Appendix A: SDS Python Code	96
Appendix B: Nmap Scan Results	109
Bibliography	118
Citations	120

Chapter 1

Introduction

1.1 Contextualization

Technology is in such a fast-moving development phase these days that the need for rapid processing and innovation has become the essence. Such a compelling motive drives the never-ending quest for progress, especially in areas like cybersecurity where the risks involved are very high. Through the interconnection of internet services and the increase of connected devices, cyberspace has turned into a major risk for organizations to deal with.

Among the various tools employed by cybersecurity professionals, network scanning techniques are essential. Apart from the security experts who use these techniques to identify vulnerabilities, assess risks, and enforce security measures to protect networks, they are also sought by malicious hackers trying to gain access to the network weaknesses.

An obvious example of how badly network scanning can be is the infamous Morris worm of 1988. This worm, created by a graduate student at Cornell University named Robert Tappan Morris, was the first to point out the vulnerabilities of the early Internet, infecting over 6,000 machines and causing an estimated \$10 million damage. This attack brought to the fore the need for robust network security measures as well as effective detection mechanisms. The detection of network scans poses enormous difficulties, especially if it is to differentiate between the legitimate activities of the network, such as vulnerability assessments, and the malicious scanning attempts.

The attackers have shifted towards more advanced tactics that enable them to bypass detection systems that are based on known attack patterns and still rely on traditional signature-based detection systems. These methods often do not cope with the complexity and changing nature of modern cyber threats.

The thesis will unfold more of the DNS-based detection methods, showcasing their efficiency in discovering and blocking possible threats. Although DNS-based detection is primary, it rather draws attention to a multi-layered security strategy where the utilization of various detection mechanisms provides an overall defense solution against cyber threats becoming more sophisticated over time.

1.2 Thesis Objectives

This thesis aims to explore and introduce DNS-based detection methods for identifying network scanning activities. Specifically, the objectives are:

- To provide a comprehensive understanding of network and port scanning techniques, including their implications for network security.
- To examine the role of Nmap as a prominent tool for network scanning, detailing its features, capabilities, and usage in security assessments.
- To introduce the Domain Name System (DNS) and its potential utility in detecting network scanning activities.
- To develop and implement a new DNS-based detection technique designed to identify network scanning activities and evaluate its effectiveness. Finally, discuss the implications of this technique for enterprise network security.

1.3 Relevance of the Research

Thesis research not only delves into the diagnostic area of cyber threats but also contributes to the body of knowledge on DNS-based detection techniques, which are relatively less focused than other elements. As a matter of fact, by presenting an innovative DNS-based detection method, this thesis is meant to improve the existing detection capabilities and create a new way of network infrastructure protection.

By using the Domain Name System (DNS), the infrastructure of internet communication, this approach can evaluate the DNS traffic patterns to find any anomalies that are presenting harmful scanning activities. This approach has the potential to provide early detection and effective mitigation of network scanning threats, enhancing overall network security.

1.4 Structure of the Thesis

The structure of this thesis is designed to systematically address the research objectives outlined above. The following chapters are included:

- Chapter 2: Network Scanning: Concepts and Techniques - This chapter defines and classifies network scanning, discussing common techniques and their associated risks. It also includes examples using Nmap to illustrate these scanning techniques.

- Chapter 3: Detection of Network Scanning - This chapter explores various methods for detecting network scanning activities, including the use of Firewalls, Intrusion Detection Systems (IDS), Intrusion Prevention Systems (IPS), and other scanning detection methods to protect a network.
- Chapter 4: DNS-based Detection of Network Scanning - This chapter begins with an overview of DNS and then delves into DNS-based algorithms for detecting network scanning activities, including the introduction and analysis of the proposed detection method.
- Chapter 5: Case Study - This chapter describes the experimental setup, including virtual machines, a custom Python script for DNS-based detection, and Nmap commands used for scanning activities.
- Chapter 6: Analysis and results - It analyzes network traffic patterns and detects malicious scanning activities using the proposed DNS-based method.
- Chapter 7: Conclusion - It summarizes key findings, provides recommendations for enhancing network security, and suggests future research directions.

Chapter 2

Network Scanning: Concepts and Techniques

2.1 Overview: why is Network Security so vital?

Can you picture the flow of information across our networks being shut down in today's world? From online banking to social media, our daily lives are intricately woven into the digital fabric. All the luxuries aside, our connected world has a big issue: the safety of our networks.

Your network may be the virtual Fort Knox, but it is still infiltrated with loopholes through which personal data (such as your financial records, private data, or a company's top-secret information) may be exposed. It is like someone opening the door, which results in a chain of bad consequences. Substantial financial losses represent some of the main negatives that can be borne, especially for businesses.

When customer data is compromised in a data breach, it can result in great damage to the reputation of the company and lead to a long period of lack of confidence.

Such an interconnected and globalized society, as it is at the moment, brings cybersecurity to the fore of all endeavors, and our dependence on computer networks for daily tasks has markedly grown.

Here are some of the primary reasons why network security is central:

- **Protecting sensitive data:** as said before, Network stores important information like financial data, personal details, and intellectual property. There are many implications if this information is stolen. It can result in a company being charged for the losses they have made and the customers being the ones whose data is bought and misused. This can also have the effect of a company losing profit as their private and important data is used illegally. Different scenarios that indicate the significance of this include, for instance, safeguarding a company's private information or, similarly, a client's credit card details, etc.
- **Ensuring business continuity:** businesses depend on their networks to function without any issues. A cyber attack can be catastrophic and can cause firms to lose money; thus, the company's reputation can also be threatened. For instance, a cyber attack can lock a company's critical systems, and hence it can stop business activities requiring a lot of money to resolve the problem.

- **Compliance with regulations:** many laws and regulations mandate organizations to implement adequate network security measures. Failure to comply can result in hefty fines and legal penalties. Regulations like GDPR, HIPAA, and PCI DSS are designed to protect data privacy and security, and non-compliance can lead to severe financial and reputational consequences.
- **Safeguarding critical infrastructure:** The most commonly attacked division of infrastructure is the government's computer networks and power grid technology systems. A country's computer networks and/or transportation system may be the target of an attack, which can subsequently result in harm to one or more segments of the society but also may provoke the destruction not only of lives but also of the surrounding infrastructure. The security of the country and the lives of its inhabitants are both dependent on the issue of the security of these infrastructures being the top priority.

Network scanning is of paramount importance in cybersecurity, as it is the key analytics tool that enables one to understand network configuration and find system loopholes. This section examines network scanning's progression through time, a formal definition, and actual applications, the last part of which indicates its mission as a key component of both defense and strategic intelligence activities.

2.2 Definition and Historical Perspective of Network Scanning

Network scanning, which is a methodical step-by-step process for discovering and enumerating active hosts as well as open ports on a system, has been the mainstay of cybersecurity since it was established in the early days of the internet. In the past, network administrators and researchers were very keen to connect with their curiosity and necessity and used some very primitive tools to find connected devices and carry out network monitoring. This process involves sending various probes or packets to network addresses and analyzing the responses to map the network topology and identify services running on connected devices. Such a proactive strategy puts the cybersecurity professionals in a position where they can apply their skills in a manner to detect threats and safeguard the network environment.

One of the earliest network scanning tools that come to mind from the late 1980s is "SuperScan" [1]. Actually, this lesser-known tool functioned ICMP (Internet Control Message Protocol) echo requests, also well-known as ping, to recognize active hosts on a network (see Figure 1). And this marked the beginning of modern network scanning methods, introducing a new era of cybersecurity that has energized and captivated professionals across the field.

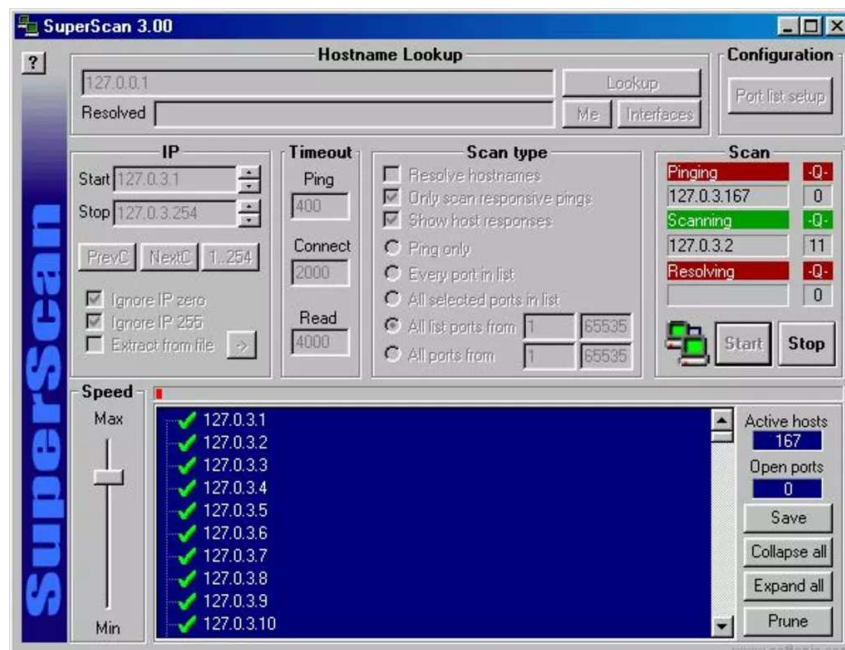


Figure 1: SuperScan tool.

1988 was the time when the history of network scans saw a very important milestone with the arrival of the famous Morris virus. This self-replicating worm, which was created by the student at the renowned Cornell University, Robert Tappan Morris, exploited vulnerabilities in Unix systems to spread across the nascent internet. Thus, the Morris worm caused an estimated 10 million dollars in damages to over 6,000 computers by infecting them over. It united everyone and reinforced the idea that vigilance and preparedness were essential.

In the wake of the Solar Sunrise incident in 1998, the need for fundamental skills like network scanning's importance in cybersecurity got proven. A group of teenagers along with one Israeli hacker were the ones to use network scanning methods that were not only sophisticated but just plain sinister. They even managed to exploit vulnerabilities they found in the U.S. military networks. By taking the help of automated tools used for Solaris operating systems, the attackers found a way to penetrate their defenses and thus were able to steal valuable sensitive data, which in turn led to widespread concern about national infrastructure security. The situation indicated the need for efficient network monitoring in order to secure critical systems.

The SQL Slammer worm was an outbreak in 2003, which was yet another incident to show the impact on network scanning. Buffer overflow vulnerability in Microsoft's SQL Server was what the worm exploited, and soon it had spread out all over the world in hundreds of servers within minutes. SQL Slammer was a kind of worm that made use of network scanning techniques to look for and infect vulnerable systems. As a result, the internet traffic was disrupted, the major financial institutions were in trouble, and the government agencies around the world were damaged.

The network scanning techniques developed as the internet grew and evolved. More advanced tools, such as Nmap (Network Mapper), which was created in 1997 by Gordon Lyon (Fyodor), emerged. Nmap has introduced various scan techniques like TCP connect scans, SYN scans, and FIN scans to disrupt and steal data from active hosts and open ports. Throughout that time, the focus of network scanning was beyond mere host identification and came to incorporate complete reconnaissance of network services, operating systems, and potential vulnerabilities. This change was definitely brought about by the increasing complexity of network infrastructures and the growing sophistication of cyber threats.

By the time the 21st century was born, network scanning had evolved (see Figure 2) and integrated automation and more advanced detection methods. Nessus [2] and OpenVAS [3] developed, allowing fully automated scans to identify network systems' security vulnerabilities. Moreover, the network scanning tools were also augmented by artificial intelligence (AI) and machine learning (ML) technologies, and their capabilities went up.

Today, network scanning is one of the most critical practices in cybersecurity. Its dual purpose, both defending and attacking information, highlights the importance of developing new detection mechanisms that can accurately differentiate between legitimate and malicious activities. It is a never-ending cat and mouse game with incredibly high stakes.

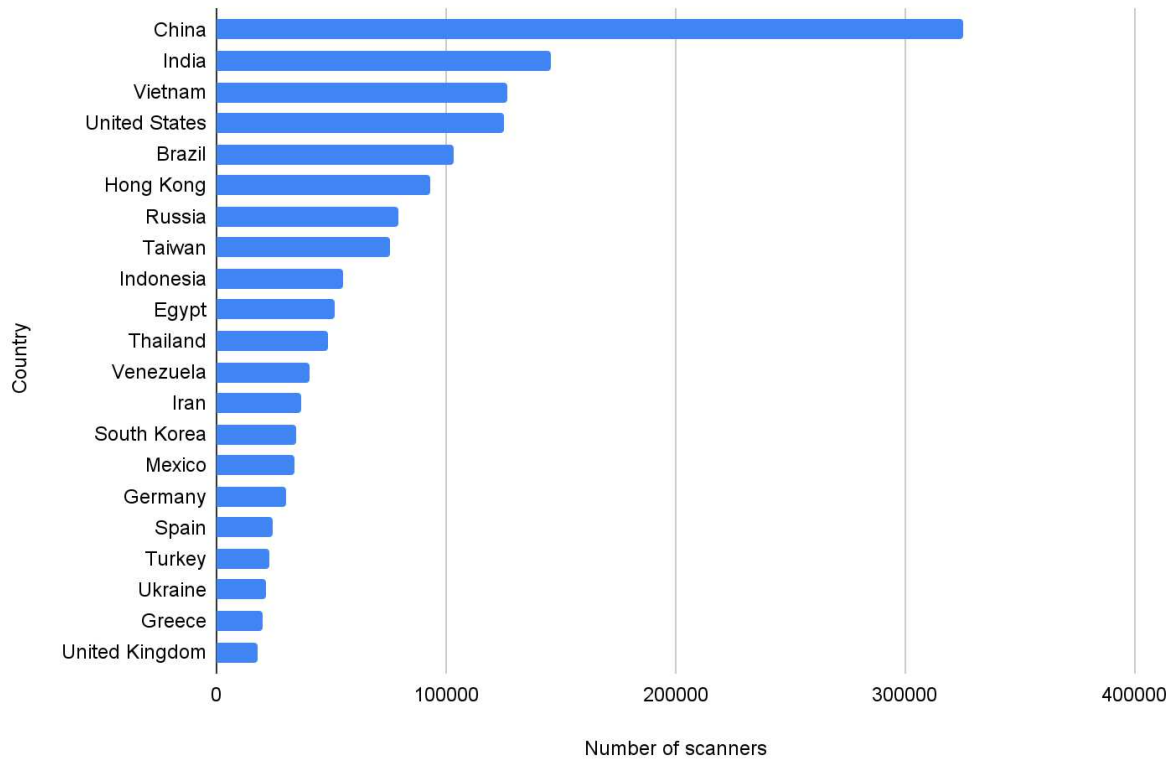


Figure 2: Top 20 countries where the largest number of scanner IPs originated. Source: Palo Alto Networks, 2021

2.3 Reconnaissance

In cybersecurity, reconnaissance techniques are employed by both defenders and attackers to gather information. Our next step in this discussion will be to detail how reconnaissance helps us understand the true essence of a secure network infrastructure while also enabling us to prevent potential threats before they can be exploited.

Reconnaissance techniques serve as the digital equivalent of scouting missions, gathering crucial intelligence to gain a comprehensive understanding of an adversary's network. Elaborating on the subject matter, reconnaissance is further decomposed into two distinct forms: passive and active. Both approaches serve distinct purposes and entail different levels of interaction with the target network.

Passive reconnaissance consists of the activities that gather information without an initial contact with systems. The primary goal of this method is to remain undetected and prevent triggering an alert to the network's defenses. Techniques include:

- Open Source Intelligence (OSINT) → it is the method that gathers the publicly available information online about organizations or individuals. Information can be accessed from websites, social media platforms, news articles, and public records. This might be the blueprint for the same type of building a business or company carried out in the near past. For example, a company's website might reveal details about its infrastructure, key personnel, and even security measures in place.
- Network Traffic Analysis → observing network traffic is one way of collecting a wealth of information. Software applications like Wireshark can trap and interpret data packets as they travel on the network, disclosing information such as the devices in use, the nature of protocols, and potential vulnerabilities. The method shown above can be very good in understanding how data is being exchanged within a network.
- Domain Name System (DNS) Harvesting → DNS records could give an impression of the framework and the hierarchy a network of a target has. Tools such as "Whois" can indeed be used to gather details about a domain, such as identifying the owner, registration, and contact dates. In addition, the DNS requests can also disclose the IP addresses of domain names that are linked to the network, and they can therefore be utilized to plot the network.
- Social Engineering → utilizing human interaction with employees or stakeholders, an attacker can reap the secret information that will then never be communicated with the computer system. Techniques like pretexting, where the attacker fabricates a legitimate

reason for asking certain questions, are often used in these scenarios.

Active reconnaissance is the sort of reconnaissance that takes place in close interaction with the system in question by sending probes or requests in order to obtain information. This methodology is more likely to be detected but may give detailed and specific information.

The landscape of active reconnaissance techniques is as diverse as the digital threats it seeks to uncover; common methods are:

- Port Scanning → it serves as a methodical technique for enumerating active ports on a target network device. This process involves sending connection requests to various ports and analyzing the responses to identify open ports and the services running on them.
- Service Identification → this process utilizes specialized tools or protocols to identify the specific services running on open ports. By pinpointing the services, security professionals gain a clearer understanding of the target's functionality and potential security risks associated with the identified software.
- Operating System Detection → by sending crafted network packets and analyzing the response characteristics, this technique attempts to fingerprint the operating system running on the target device.
- Vulnerability Scanning → it automates the process of identifying known weaknesses in software, configurations, and network devices. Utilizing a database of vulnerabilities, there are tools that scan target systems to identify potential security flaws.
- Banner Grabbing → it is a swift reconnaissance technique that exploits the initial communication handshake between a client and a server. During this handshake, some servers transmit a banner message that reveals information about the service running on the port.

Furthermore, through these two major types, we can differentiate two more methods of reconnaissance based on the scope: target-specific reconnaissance and wide-range reconnaissance.

Target-specific reconnaissance entails intense and stealthy scanning techniques that are concentrated on the collection of the entire information related to a host or a network without triggering any red flags. This detailed method in turn involves a multi-phased process. One tactic uses a divide-and-conquer strategy with separate systems for scanning and attacking. For instance, imagine a detective using a burner phone for initial recon (scanning) and then switching to his own phone (attacking) to avoid linking the activities. The division of responsibilities is achieved mainly by using compromised systems called throwaway systems as scanning platforms, which makes it extremely difficult to find back the offending source. In fact, tracing efforts in such cases typically lead back to innocent third parties, further obfuscating the attacker's true

identity.

Another technique takes advantage of the power of numbers: distributed scanning utilizes multiple compromised systems (think a network of informants) to scan different parts of the target network simultaneously, significantly reducing the scanning footprint from any single system and thereby minimizing the likelihood of detection. A central node often collects and consolidates the scanning results, providing a comprehensive view of the target's vulnerabilities. This method is particularly effective when utilizing botnets, which can execute coordinated scans across thousands of compromised systems, making detection extremely difficult. A botnet, consisting of several compromised systems (bots), is deployed to scan. Each bot sends an implicit single packet to the target, creating a distributed and nearly untraceable scanning campaign.

Lastly, the ability of an attacker to scan a target is done by scanning without sending any packets from their own computer to the network. The attacker uses a third-party system (an idle host) to do the scan instead. When an illegal scan is performed on the network, no primary system will be used to point out the location of the culprit. On the contrary, the target system will return data to the idle host. First, the idle host is the one getting the returned data, which is the information that the attacker sought. IT professionals call it idle scanning. We will be back to this topic in the near future while discussing the port scanning in detail. Also, the criminals may send the scans very slowly over a long period of time to escape detection. This technique, "low and slow", is the strategy for criminals to mix the method with regular network traffic, and in this way, they circumvent the detection. By spreading the scanning activity over days, weeks, or even months, attackers minimize the chances of their actions being flagged as suspicious.

In contrast, specifically designed to operate over a large area, the **wide-range reconnaissance** is likened to a search party that casts out a digital net in order to find out the existence of this service and sees a particular weakness or whether it is spread across the region. In this case, speed and efficiency are key - automated tools are widely used to mitigate the need for humans and, in turn, automate the system that enables it to become very efficient in collecting data (less human labor is employed).

Masscan [4] and ZMap [5] prove the point and are meant to scan every single IP address in a very short time. The elaboration made by the tool heavily depends on its goal, and, therefore, purpose-built weapons featuring auto-rooters, which are highly specialized tools, actually act as one-shot scanning combined with automated attack. The auto-rooter instantly launches an exploit, attempting to compromise the vulnerable system once an open port is detected. The modus operandi involves inadvertently including vulnerable systems in a botnet.

Furthermore, scanning worms usually take over all systems without making a distinction. They,

being of the self-replicating type, reproduce on their own, in truth, as if they were digitally infected with viruses that propagate explosion-like across a specified network by exercising known vulnerabilities. An infected system is then set to scan for other courses of action by means of ensuing effects that compel the irrevocable collapse of undeniable traditional fortifications. Reconnaissance, in short, plays the main role in the initial part of the simulation of cyber-attacks, and it is a process that normally goes before hardening the system. In fact, penetration testing (or ethical hacking) mainly comprises three stages: information gathering, reconnaissance, and network scanning (see Figure 3).

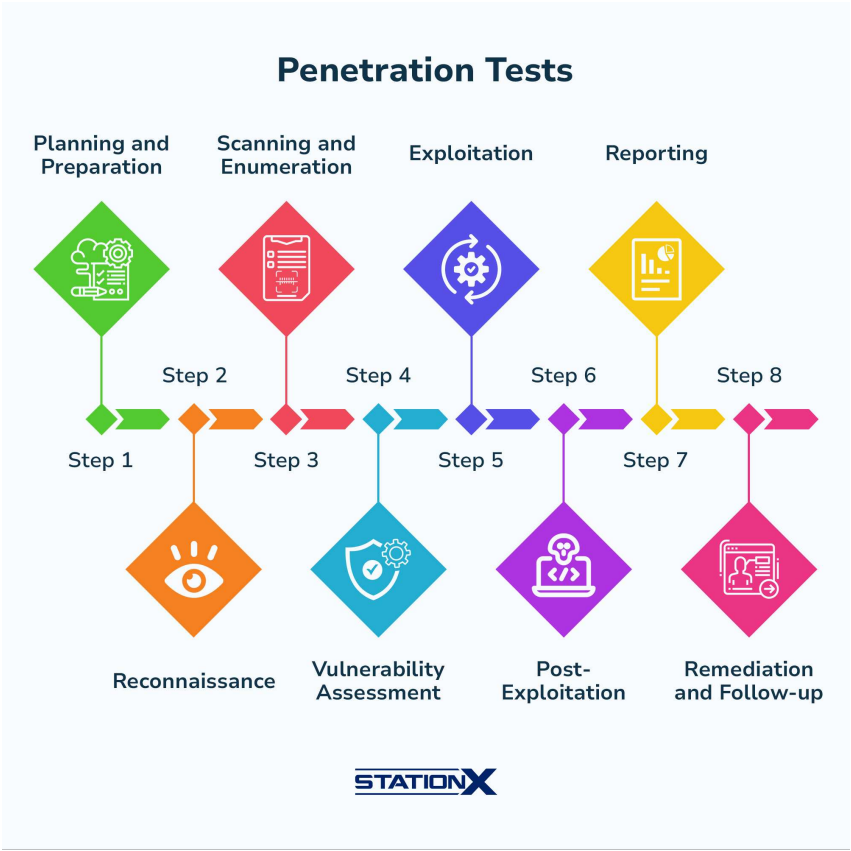


Figure 3: Penetration test methodology. Source: STATIONX

Information gathering starts the process by where the examination teams begin to collect as much information as they can on the target. Passive reconnaissance techniques are often used. A few methods called internet searches and DNS analysis, along with looking at the public records and social media, are always found on the list. In fact, DNS records can be exposed to secrecy concerning the network's structure and hierarchy as a consequence of analyzing them.

The network scanning process, which we will expand on in detail later, is the concluding step in the initial prep activities leading to an actual penetration effort. Through this stage, hackers are able to not only identify the potential areas to breach, but also find out the normal traffic patterns in the network, which are very useful to detect the anomalies during the practical simulation of an attack.

2.4 Network Scanning Strategies

In a corporate network, it is essential to first assess the number of targets and the direction of the scanning activity. This systematic approach aids in developing more effective security strategies and facilitates the detection of potential threats. There are three major types of network scans, based on direction:

- **Local-to-local scans:** the scans of this kind are the ones that are produced and consumed on the same devices in a specific local area network (LAN) segment. Such scans are most frequently used by network administrators to perform such functions as finding printers or servers, resolving connectivity issues, managing the network assets, etc. (see Figure 4).

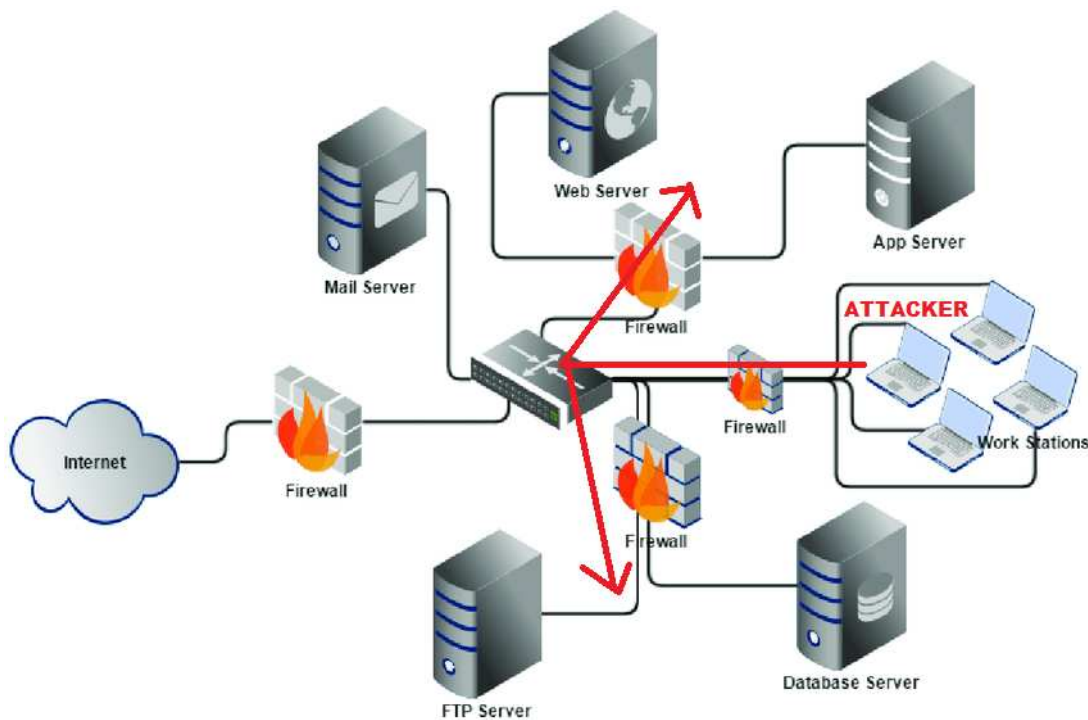


Figure 4: Local-to-local scan.

Besides, security compliance is also a major use of local-to-local scans, and in fact, administrators can oversee all devices to make sure that they meet the organization's security policies and standards and to find unauthorized devices that might have connected to the network without proper approval. Identifying such unauthorized connections is vital for maintaining the integrity and security of the network!

Moreover, the vulnerability assessment component of the local-local scanning tools is indispensable, as it helps identify weaknesses within the local area network. For instance, it can detect systems lacking patches or devices that are improperly configured, thereby

reducing the likelihood of these vulnerabilities being exploited by attackers. The integral part that performance also represents is as follows: the network administrators become aware of the utilization choke points and eliminate them by the recourse to better configurations for efficiency improvement. Where it is most often the case of scanning devices that are benign, the presence of unauthorized local scans can signal the presence of malware entities or subverted devices in the network that are trying to map out this local network. These incidents present the case of the need to control and observe periodically the scanning being applied to the networks to ensure no security breaches will be exploited.

- **Local-to-remote scans:** those are the instances when the device within the LAN initiates a scan that is targeting a system that is located somewhere outside the local network, often on the internet. Such scans typically involve a security professional doing a vulnerability scan on a web server or a desktop user accidentally scanning an outside device besides a web server (see Figure 5).

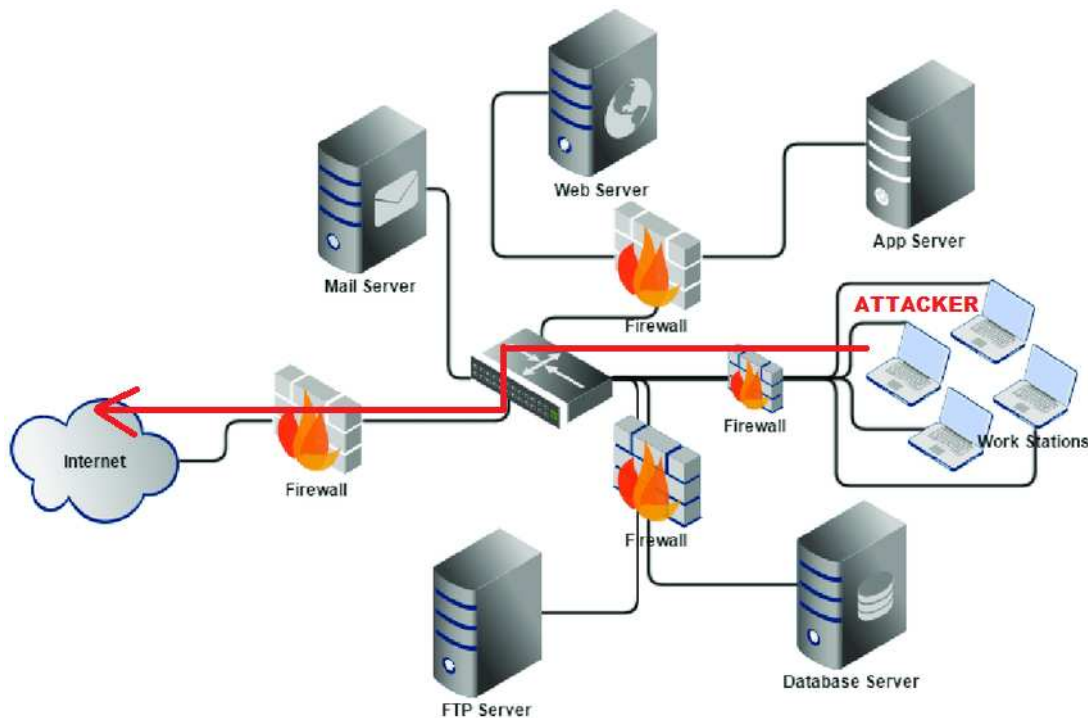


Figure 5: Local-to-remote scan.

These scans are serving functions that are essential both in network management and security. As an example, security professionals could be running vulnerability scans on web servers that are outward facing to pin point and ultimately fix potential weaknesses that are being targeted by intruders. Furthermore, the scans could be used on external servers to see if their compliance with their policies is maintained and if the industry standards

are followed.

Besides this, the system administrator can also perform scans on external servers and services to check their rich accessibility performance. This is very essential for the organizations that depend on cloud services or remote data centers.

Nevertheless, teared-up or unauthorized outbound scans are likely to cause the data leakage by accident or may even be considered as malevolent behavior by the target; thus, security defenses and legal issues could be initiated.

So, the foremost thing we need to do is come up with rigid policies and mechanisms for surveillance that will mitigate the potential damage either on purpose or intentionally while scanning from one location to the other.

- **Remote-to-local scans:** this expression is given to scans that come from a device external to the enterprise network and are directed to systems located in the network. Scans of this type are at the top of the list of security professionals' anxieties because they might be signs of a hacker trying to find weak or vulnerable parts of the network's perimeter defenses (see Figure 6).

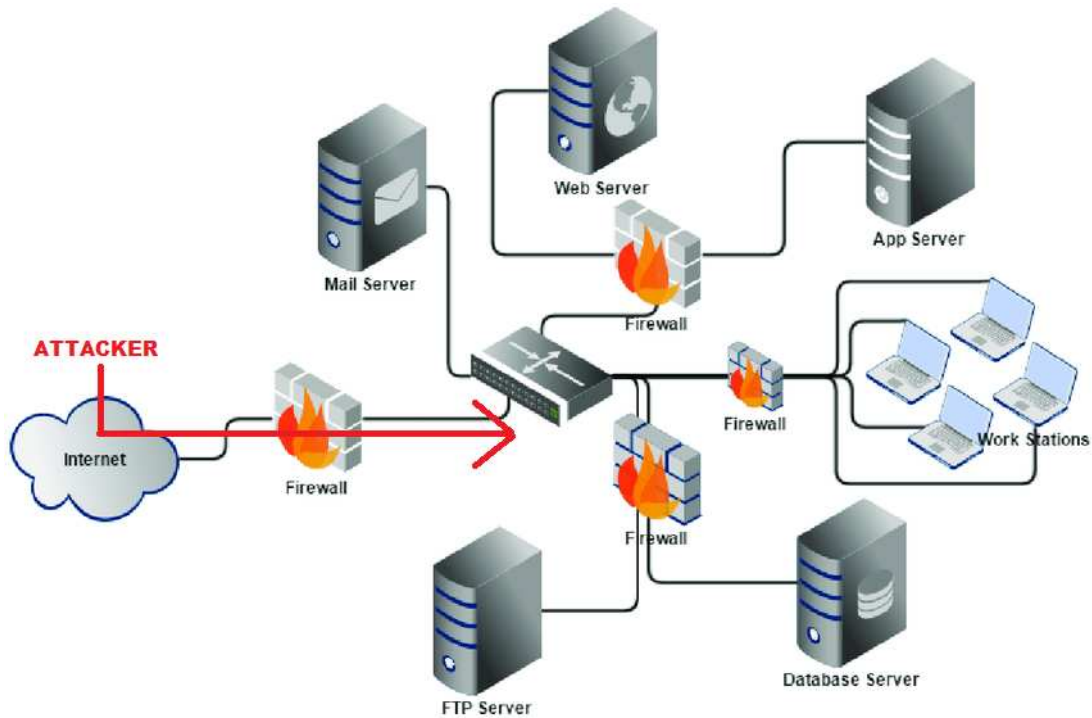


Figure 6: Remote-to-local scan.

Network administrators, on the other hand, should also be careful to spot and analyze remote-to-local scan activity to ensure the security of the network. The efforts of such devices as firewalls, intrusion detection systems (IDS), and intrusion prevention systems (IPS) are instrumental in the identification and blocking of these unauthorized scanning activities. It is undoubtedly essential to carefully monitor and respond quickly when a system is not functioning as expected.

Nevertheless, remote-to-local scans are not exclusively malicious; they can also be a legitimate section of the security assessments conducted by entrusted external entities. One widespread example for this is to overview safety affairs, due to which the auditors perform these scans to scrutinize networks' face-side security.

Among the various network scanning tools available, one of the most powerful and widely-used is **Nmap (Network Mapper)**.

NMAP ("Network Mapper") is a free & open source utility for network discovery and security auditing. [6].

Learning the way Nmap finds information through scanning includes the details of how it interacts with network protocols and interprets responses from target systems.

Nmap basically functions on three main network protocols, each of them having their own specific uses and purposes:

- **TCP (Transmission Control Protocol):** it is a connection-oriented protocol that establishes a connection between devices to ensure reliable data transfer. It is highly suitable for port scanning as a connection is established where combustion of very detailed information regarding the target service is possible. Flags such as SYN, ACK, FIN, RST, and so on are very important in various scanning methods using TCP.
- **UDP (User Datagram Protocol):** it is a connectionless protocol that sends packets without establishing a connection. This makes UDP scans fast but less reliable because there are no acknowledgments. UDP scans are often used to identify open UDP ports and associated services.
- **ICMP (Internet Control Message Protocol):** it is primarily used to send a message to the destination and for diagnosing purposes, host discovery, and certain types of network probing. For example, ICMP echo requests (ping) are commonly used to check if a host is alive and responsive.

In particular, Nmap excels in five key areas: discovering hosts, scanning ports, detecting services and their versions, identifying operating systems, and utilizing the Nmap Scripting Engine

(NSE). In the upcoming paragraphs, we will delve into each of these capabilities (except NSE) to highlight their importance in evaluating network security.

2.5 Host Discovery

Host discovery is one of the fundamental functions of Nmap, allowing us to determine if a target host is online and responsive. This function is crucial for focusing subsequent scans only on active hosts, making the scanning process more efficient and relevant.

Nmap employs various methods for host discovery, each tailored to different network environments and scenarios:

- **ARP Ping (for Local Networks):** when Nmap uses ARP Ping, it sends out special requests called ARP probes to check every IP address within a specified range. Each device on the local network that has the target IP address will respond by sharing its unique MAC address. This response confirms that the device is active and connected to the network, since ARP is crucial for linking IP addresses to specific devices on a LAN. To perform ARP ping with Nmap, we can use the command:

```
nmap -sn <target>
```

- **ICMP Echo Request (Ping):** this method is used to check if a device is connected and responding on a network. When Nmap sends a Ping, it is like asking the device, "Hey, are you there?". If the device is online and configured to respond, it will send back a reply saying it is ready to communicate. Therefore, it is very useful for identifying which devices are active on a network.
- **ICMP Timestamp Request:** this method is a lesser-known but useful tool in network diagnostics because it checks if devices are online (just like Ping) and gets the device's time, which can help with syncing clocks and diagnosing time-related issues.
- **TCP SYN Ping:** by sending a SYN (synchronize) packet to a specific port on the target device, Nmap uses this technique to check if a device is online, and if it is and the port is open, the device replies with a SYN-ACK (synchronize-acknowledge) packet. To use TCP SYN Ping with Nmap, we can use this command:

```
nmap -PS<port> <target>
```

Obviously, we can specify the port we want in the command above, for example, the port 443 (in this case, it would be: `nmap -PS443 target`).

- **TCP ACK Ping:** Nmap sends a TCP ACK (acknowledge) packet to a specific port on the target device, and if it is online, it usually responds with an RST (reset) packet, indicating that the device received the packet but was not expecting it (which confirms

the device is there).

TCP ACK Ping Nmap command:

```
nmap -PA<port> <target>
```

- **UDP Ping:** it is like sending a quick message to see if anyone responds. Unlike TCP, UDP does not establish a connection, so the response behavior can be different. Hence, when Nmap sends a UDP packet to a port (on a device), if the port is open and the device is up and running, you might receive a response from the application listening on that port; otherwise, the device typically sends back an ICMP "Port Unreachable" message, confirming that the device is online but the port is not open.

UDP Ping Nmap command:

```
nmap -PU<port> <target>
```

- **IP Protocol Ping:** Nmap sends packets using various IP protocol numbers, such as IGMP (which manages multicast groups) or other specific protocols. This technique is particularly useful in environments where traditional ping methods are blocked or filtered. A practical example could be:

```
nmap -PO1,2,4 192.168.1.1
```

In this example, Nmap sends packets using IP protocol numbers 1 (ICMP), 2 (IGMP), and 4 (IP-in-IP) to the target IP address (192.168.1.1).

- **Combination of Methods:** using a combination of host discovery methods in Nmap enhances the accuracy and comprehensiveness of network scans. Indeed, different devices and networks may respond to different types of probes, thereby enabling the detection of as many active hosts as possible and increasing the likelihood of reaching devices behind defenses (bypassing firewalls and other security systems). In Nmap, we can use options that include multiple types of pings in a single command like this:

```
nmap -PE -PS80,443 -PU123 -PA22 -PP 192.168.1.0/24
```

In this example, Nmap will send ICMP Echo Requests (-PE), perform TCP SYN Pings on ports 80 and 443 (-PS80,443), send UDP Pings on port 123 (-PU123), send TCP ACK Pings on port 22 (-PA22) and use ICMP Timestamp Requests (-PP) against the target.

2.6 Port Scanning

Port is a communication endpoint in a computer network. It is a place in the network where a system allows the passing of data to the Internet between the devices. Ports, associated with IP addresses, represent specific processes or services on a device. Each port is assigned a unique number, ranging from 0 to 65535, called the port number. Ports are useful for enabling a variety of applications and applications to run together on the same device without interfering with each other. They can be divided into three types: well-known ports (0-1023), registered ports (1024-49151), and dynamic/dedicated ports (49152-65535).

Port scanning is the method of sending packets to different ports on a network device and then analyzing the responses to decide the status of those ports, give a deep look at their services operating on the machine, their version numbers, and most importantly, if these services are susceptible to some form of an attack. Port scanning is a vital step in the process of finding the services and potential vulnerabilities of the device.

There are three primary port states that a scanner can identify:

- **Open:** the port is active, and a service is listening for connections.
- **Closed:** the port is reachable but not in use.
- **Filtered:** the port is protected by a firewall or other security measure, making it difficult to trace its status.

Specifically, Nmap divides ports into six states: open (an application is actively accepting TCP connections, UDP datagrams or SCTP associations on this port), closed (the port is accessible and it receives and responds to Nmap probe packets, but there is no application listening on it), filtered (Nmap cannot determine whether the port is open because packet filtering prevents its probes from reaching the port), unfiltered (the unfiltered state means that a port is accessible, but Nmap is unable to determine whether it is open or closed), open|filtered (Nmap places ports in this state when it is unable to determine whether a port is open or filtered), or closed|filtered (this state is used when Nmap is unable to determine whether a port is closed or filtered).

Considering the target selection strategy and the chosen ports, we can classify port scans into four primary groups:

- **Horizontal scanning:** it involves scanning the same port across multiple IP addresses. This technique is often used to identify a specific service running on a large number of hosts within a network. For instance, a horizontal scan might target port 80 (HTTP) across a range of IP addresses to find all web servers within a network.
- **Vertical scanning:** it focuses on scanning multiple ports on a single IP address, and it is useful for gaining a comprehensive view of all services running on a particular host.

- **Block scanning:** it combines elements of both horizontal and vertical scanning, targeting a range of ports across a range of IP addresses (typically used in large-scale network assessments).
- **Strobe scanning:** it is a targeted approach that focuses on specific, commonly used ports.

Several port scanning techniques exist, each with its own unique approach. Understanding these techniques helps in both conducting thorough security assessments and defending against potential attacks. Port scanning techniques vary based on the method of probing and the type of information sought:

- TCP Connect Scan: it establishes a complete TCP connection with the target port by completing the three-way handshake process. In this method, the scanner initiates the process by sending a SYN packet to the target port. If the port is open, the target responds with a SYN-ACK packet. The scanner then sends an ACK packet to finalize the handshake and immediately closes the connection by sending an RST packet (see Figure 7). It is particularly advantageous in non-privileged mode, as it does not require root privileges to operate. However, it is easily detectable by target systems and logged in their event logs since it completes the full handshake.

Nmap example command:

```
nmap -sT <target>
```

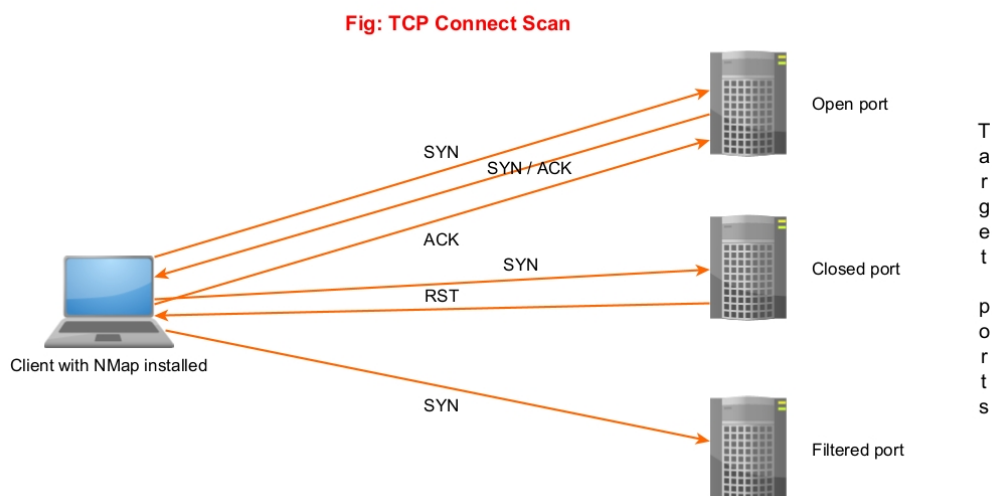


Figure 7: TCP Connect scan.

- SYN Scan: (also known as half-open scanning) it is one of the most popular and efficient techniques used in port scanning. This method takes advantage of the TCP three-way handshake process to identify open ports without completing the full connection, making it stealthier (a preferred choice for reconnaissance) and less likely to be logged by the target system (see Figure 8). This is how it works:

1. The scanner sends a TCP SYN packet to the target port.
2. If the port is open, the target responds with a SYN-ACK packet; otherwise, if the port is closed, the target responds with an RST (reset) packet.
3. Upon receiving the SYN-ACK, the scanner sends an RST packet to terminate the connection before the handshake is completed.

Nmap example command:

```
nmap -sS <target>
```

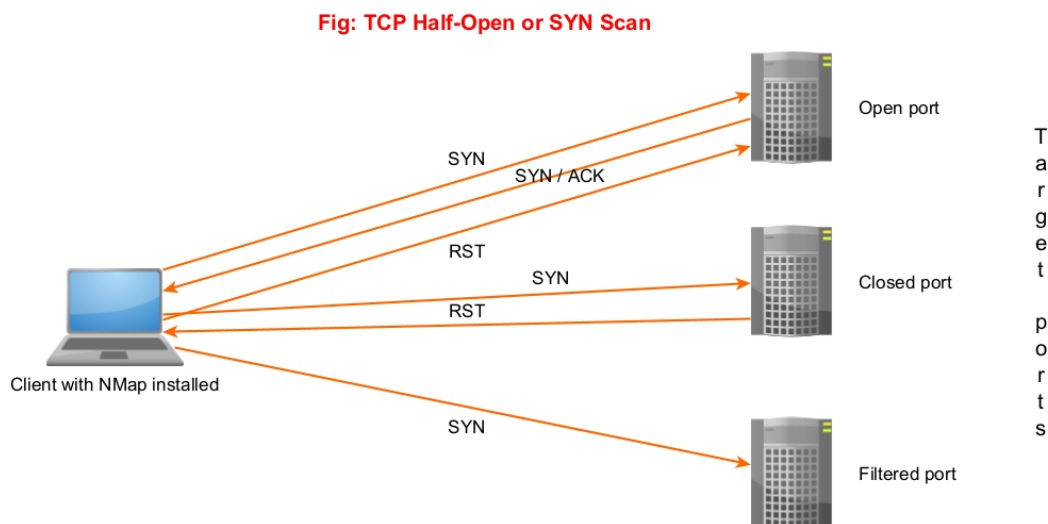


Figure 8: TCP SYN scan.

- FIN Scan: it is another stealthy technique used in port scanning that exploits the behavior of TCP/IP stack implementations. This method is effective for evading firewalls and intrusion detection systems, as it does not attempt to establish a full connection with the target port. The effectiveness of FIN scans largely depends on how the target system implements the TCP/IP stack, which follows guidelines set by Request for Comments (RFC) documents. Specifically, RFC 793 outlines how TCP should manage various packets, including FIN packets. However, because it does not explicitly define the handling for open

ports, different operating systems may implement this in varied ways. For instance, many Unix-based systems, including Linux, adhere closely to RFC 793, making their responses to FIN scans more predictable, while Windows systems might have different behaviors, sometimes not responding to FIN packets for both open and closed ports (see Figure 9). Here is how it works:

1. The scanner sends a TCP FIN (finish) packet to the target port, which typically signals the end of a connection, even though no connection was previously established.
2. If the port is closed, the target system sends back an RST (reset) packet, indicating it does not recognize the unexpected FIN packet. Otherwise, if the port is open, the target does not respond.

Nmap example command:

```
nmap -sF <target>
```

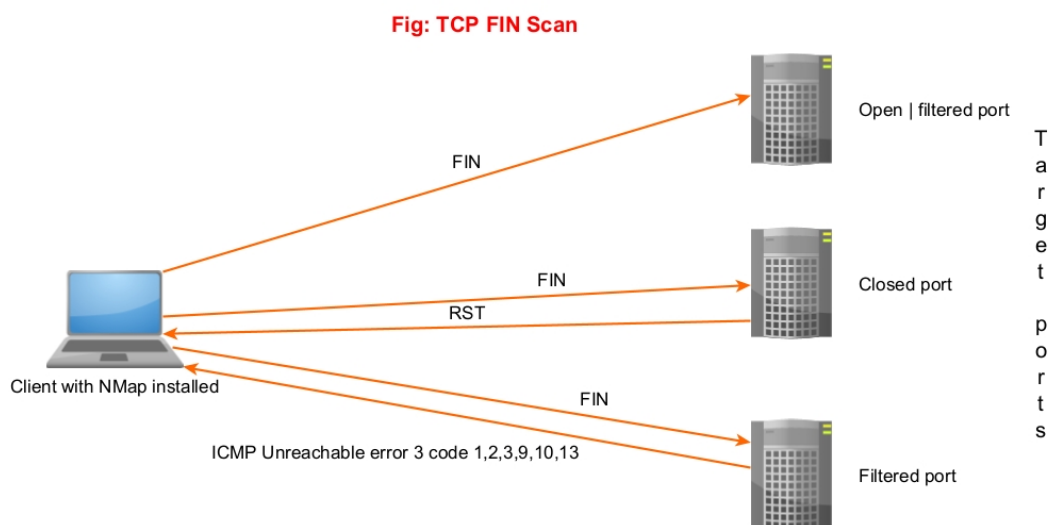


Figure 9: TCP FIN scan.

- Null Scan: it is another stealthy technique used in port scanning to determine the state of a port on a target system. Unlike other scan types, a NULL scan sends packets with no flags set (TCP flag header is 0), creating a packet devoid of any control flags (see Figure 10). Similar to the FIN scan, NULL scans are particularly useful for bypassing firewalls and packet filters that may not log or block this type of traffic, but the effectiveness of NULL scans depends significantly on how the target system implements the TCP/IP stack (RFC 793).

This is how a NULL scan operates:

1. The scanner sends a TCP packet with no flags set to the target port.
2. If the port is closed, the target system responds with an RST (reset) packet; if not, the target system does not respond.

Nmap example command:

```
nmap -sN <target>
```

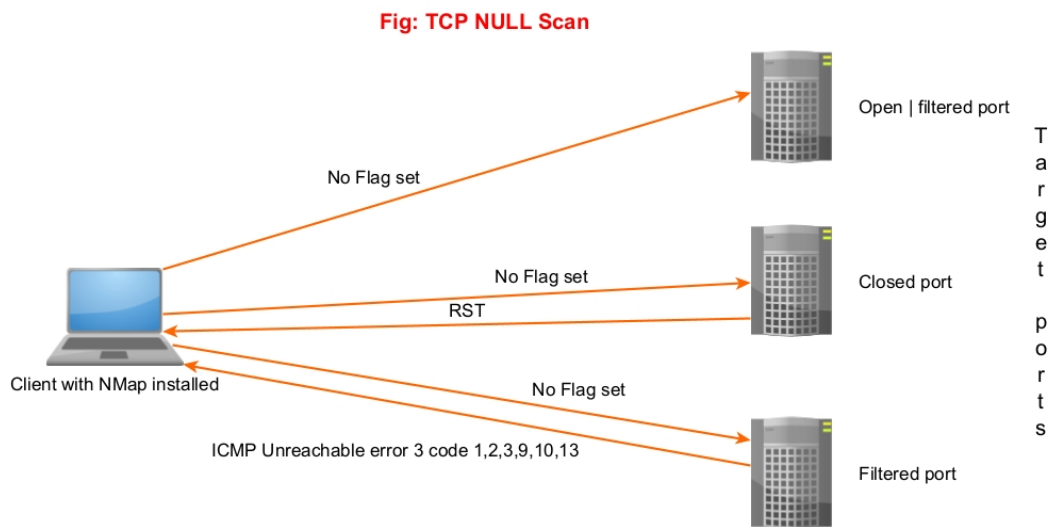


Figure 10: TCP NULL scan.

- Xmas Scan: it employs a more cryptic approach to identify open and closed ports on a target network: it sends a packet with a peculiar combination of FIN, URG, and PSF flags, lighting the packet up like a Christmas tree (see Figure 11). The unusual combination of flags might sometimes evade basic firewall rules designed to detect traditional scans; however, it is hard to know if a port is open or closed because the answers are all messed up.

This is how it works:

1. The scanner sends a packet with the FIN, URG, and PSF flags set to each chosen port on the target machine.
2. The behavior on open ports can vary depending on the target system. In some cases, no response may be received, while in others, an RST (reset) packet might be sent. Instead, in case of closed port, they will usually send back a clear message (TCP RST packet).

Nmap example command:

```
nmap -sX <target>
```

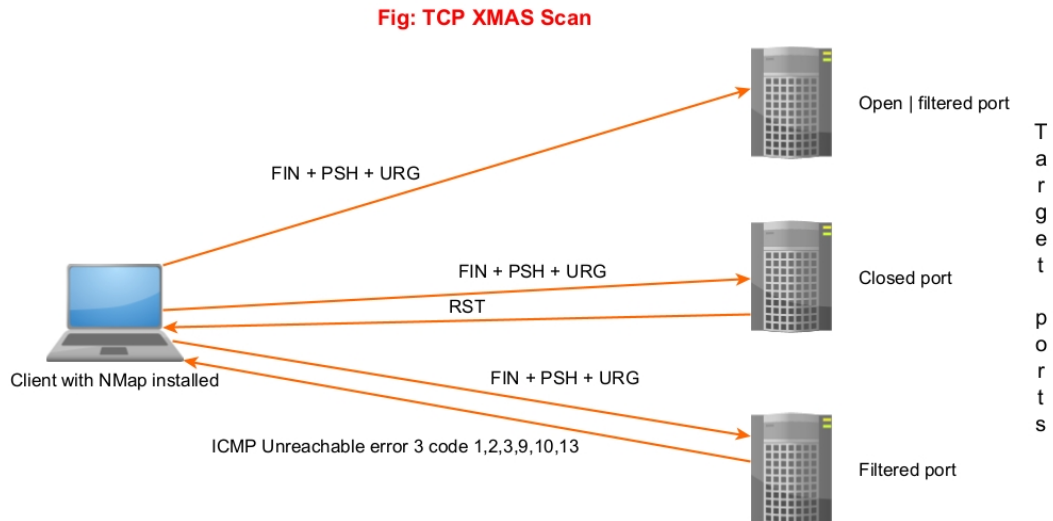


Figure 11: TCP XMAS scan.

- ACK Scan: it is another type of stealthy port scanning technique, and, unlike SYN, FIN, or NULL scans, an ACK scan employs ACK (acknowledgement) packets to probe potential ports. Although it may not provide definitive answers about open or closed ports, it offers valuable insights into firewall configurations and can be particularly useful in certain scenarios (see Figure 12).

How an ACK scan works:

1. The scanner sends a TCP packet with the ACK (acknowledge) flag set to the target port.
2. If the port is filtered, no response or an ICMP "destination unreachable" message indicates that a firewall is blocking the port. Alternatively, if the port is unfiltered, the target responds with an RST (reset) packet, regardless of whether the port is open or closed.

Nmap example command:

```
nmap -sA <target>
```

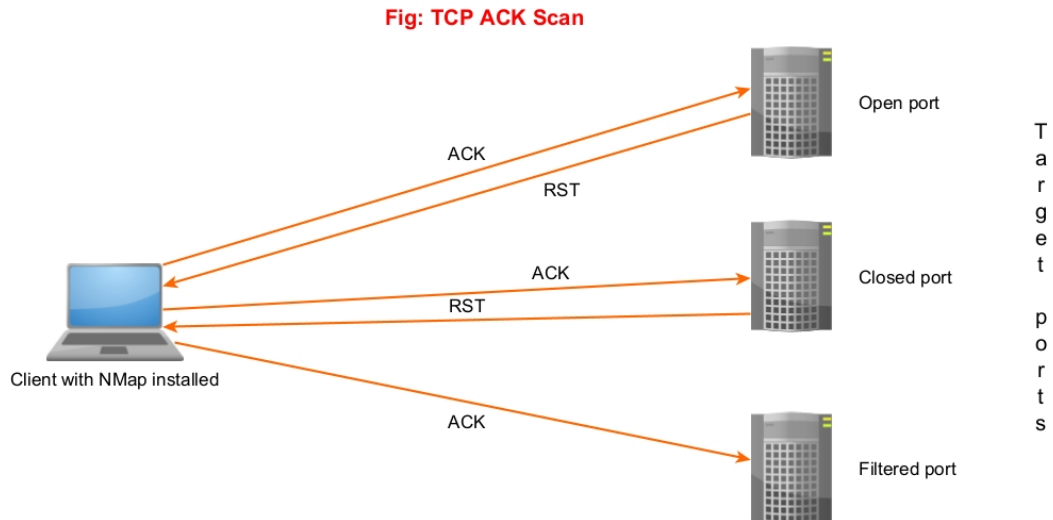


Figure 12: TCP ACK scan.

- PSH Scan: it is a relatively uncommon port scanning technique, and it utilizes the TCP PSH flag, which instructs the receiver to process the data immediately rather than buffering it. A PSH scan can be useful to identify open ports or to bypass specific firewall rules.

How a PSH scan works:

1. The scanner sends a TCP packet with the PSH flag set to the target port.
2. If the port is closed, the target responds with an RST (reset) packet, else the target might not respond at all, depending on how the TCP/IP stack of the target system handles the PSH flag.

Nmap example command:

```
nmap -sF --scanflags PSH <target>
```

- Window Scan: it is a specialized technique used to determine the state of a port by analyzing the TCP window size in the response packet. By sending an ACK packet and examining the window size in any resulting RST packets, it is possible to infer whether a port is open or closed. A Window scan works as follows:

1. The scanner sends a TCP ACK packet to the target port and, unlike an ACK scan, the Window scan examines the TCP window size field in the RST packet.
2. In the case of a closed port, the target typically responds with an RST packet with a window size of zero; otherwise, the target might respond with an RST packet with a non-zero window size, indicating the port is open.

Nmap example command:

```
nmap -sW <target>
```

- Idle Scan: it is an advanced and stealthy port scanning technique used to assess the state of a port on a target system without revealing the scan source. This method involves using a third-party system, known as a "zombie", to send packets to the target. By monitoring the responses of the zombie system, the scanner can determine the status of the target ports (see Figures 13, 14, 15).

How the IDLE scan works:

1. The scanner first identifies a suitable zombie host, which is a system with an incrementing IP ID field. This field is crucial because it allows the scanner to track the zombie's response packets.
2. The scanner sends a SYN/ACK or other probe packet to the zombie and records the IP ID value from the response.
3. The scanner then sends a spoofed SYN packet to the target, pretending to come from the zombie. If the target port is open, the target responds to the zombie with a SYN/ACK. If the port is closed, the target responds to the zombie with an RST.
4. The scanner probes the zombie again and checks the IP ID value. If it has incremented by two, this indicates that the zombie sent an RST packet in response to an unsolicited SYN/ACK from the target, meaning the target port is open. If the IP ID incremented by only one, it suggests that the zombie did not send an RST packet, indicating that the target port is closed.

Nmap example command:

```
nmap -sI <target>
```

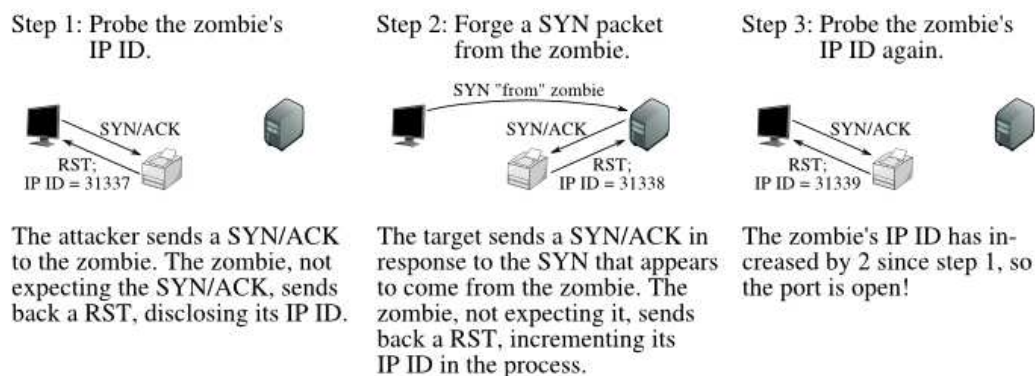



Figure 13: TCP Idle scan (open port). Source: nmap.org

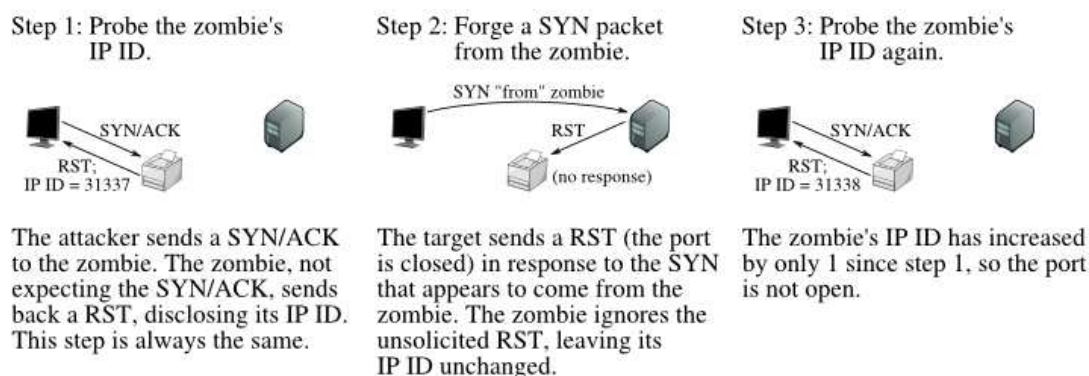


Figure 14: TCP Idle scan (closed port). Source: nmap.org

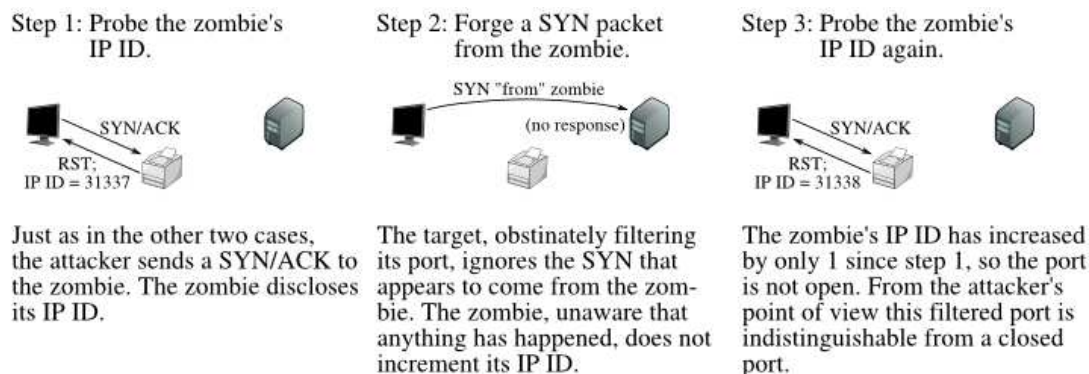


Figure 15: TCP Idle scan (filtered port). Source nmap.org

- TCP FTP Bounce Scan: this technique utilizes the FTP server's capability to connect to other servers, effectively using it as a proxy for scanning other systems indirectly. This means, Nmap sends specially crafted FTP commands (to the FTP server under scan) that instruct the FTP server to connect to other systems on the network, exploring specific ports or services of interest during the scan.

- UDP Scan: Unlike TCP, UDP is connectionless, and this characteristic makes UDP scanning more challenging because responses might be limited or not returned at all. When Nmap sends UDP packets to a range of ports on the target system, each port might respond differently: if a UDP port is open, the target system might respond with an ICMP port unreachable message, an application-specific response, or no response at all. If the port is closed, an ICMP port unreachable message is typically received. While, a filtered UDP port means that a firewall or other network filtering device is blocking the UDP packet and thus no response is received (see Figure 16).

Nmap UDP Scan example command:

```
nmap -sU <target>
```

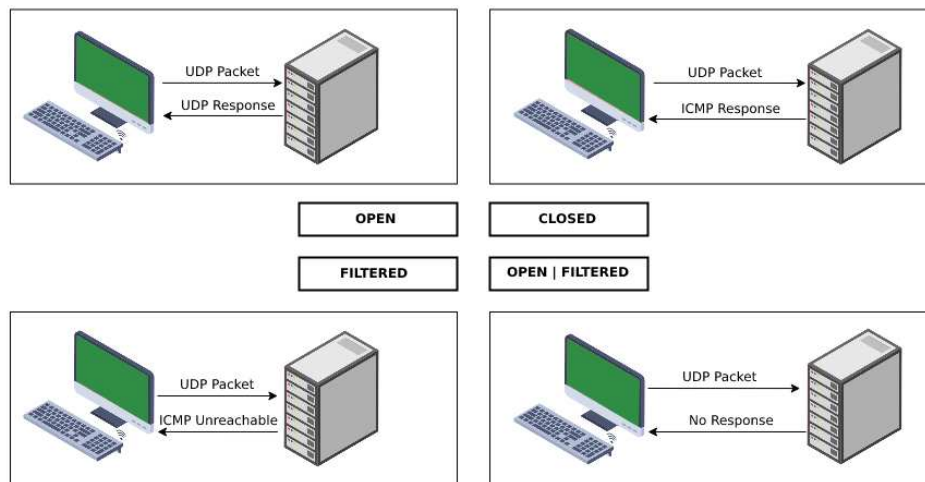


Figure 16: UDP scan.

2.7 Service, Version and OS Detection

The process of identifying which computers on a network are active and the ports they have open is very important, although Nmap goes beyond discovering the software's hidden surface through its version and service determination features. This makes it possible for us not only to tell "who" is present but also "what" they are running.

The knowledge of the software versions is beneficial in evaluating the potential risks of security breaches; the fact that the software is old can mean that it has security holes that the intruders could exploit. Nmap manages to do this through the creation of targeted packets that provoke certain services to send back the replies. As a result, by analyzing these responses, Nmap can often pinpoint the specific service (like a web server or email server) and even the software version it is running. In fact, Nmap compares the observed responses against its extensive version detection database (or service fingerprint database), in order to match detected responses to specific software versions.

Moreover, Nmap's version detection goes beyond simple identification by providing detailed information about protocols and options supported by each service, such as uncovering SSL/TLS versions, HTTP headers, supported methods, and other protocol-specific details.

To perform a basic version detection scan, it is possible to use this command:

```
nmap -sV <target>
```

In addition, Nmap offers powerful OS detection capabilities that enable Nmap to determine the operating systems of target hosts by detecting nuanced differences in their network responses and behaviors.

Nmap command for OS detection:

```
nmap -O <target>
```

Chapter 3

Detection of Network Scanning

3.1 What is Detection?

Network scanning detection is the primary element of the network security maintenance, as well as the most critical one. When scanning activities are detected at an early stage, organizations can prevent intrusions before an attack is even planned.

Specifically, network scanning detection is the procedure through which detection systems identify and analyze attempts to gather information about a computer network. This is related to the monitoring of network traffic for patterns and activities that indicate someone is trying to determine if your network has active devices or unsecured ports or plans to find and use vulnerabilities. Sophisticated detection approaches may alert the network administrators to the threats that they may encounter in the future, thus, such data will aid them in implementing protective steps.

Attackers are always creating new scanning techniques to bypass detection methods. Network defenders are to get transparent about risks and how they manage them. They must keep updated with the most recent trends and transform their strategies as per the requirement. This battle of wits demands always being on alert and can normally be resolved through a proactive approach towards network security. Finding a certain IP address and refusing it minimized the effectiveness of many attacks in the network security because the attacker did not manage to establish the connection. However, it is quite important to find out the false results where a legitimate network activity is highly suspicious. This can result in time wasted on the detection of harmless events. By adjusting the detection rules and incorporating the advanced analysis algorithms, we can significantly cut down on the number of false positives and make the system more accurate.

Moreover, not all network scans are malicious. In fact, administrators will often use scans for their everyday operations or troubleshooting of their computers to check for problems. It is vital to divide these between the malicious and normal scans. The main reason for this is to develop the proper countering techniques. This might involve context-aware analysis and, in some cases, manual intervention to understand the purpose of the scan.

While detection is an important part of security, following strong network security guidelines can bring down the rate of successful breaches. Firewalls, appropriate access control, network seg-

mentation, and regular vulnerability assessments can create a more secure environment. They can serve as a deterrent for the attackers from being able to carry out effective scans as well as reducing the possibility of cyber attacks in general.

To effectively detect scanning attempts, network protectors offer a multitude of tools. Their weapons are as follows: **Firewalls**, **Intrusion Detection Systems (IDS)**, **Intrusion Prevention Systems (IPS)** and **Network Traffic Analysis (NTA)** tools.

3.2 Internal VS External Detection of Network Attacks

The detection of network scanning or attacks can be commonly divided into internal and external detection, with each of them concentrating on different types of threats and thus proffering a number of dissimilar techniques and tools for ensuring the network infrastructure.

Internal detection focuses on finding dangers happening inside the network. These dangers can come from users who are falsely promoted to more elevated positions, internal devices that have been compromised, or from lateral maneuvering by the intruders who have already circumvented the network perimeter. The purpose of network monitoring and the investigations behind it is to surveil internal network traffic and identify potentially malicious activities, such as attacks or reconnaissance attempts. One of the main methods of internal detection is Network Traffic Analysis (which we will describe later in this chapter).

Intrusion Detection Systems (IDS), Intrusion Prevention Systems (IPS) and behavior-based detection techniques are also particularly effective for internal threats.

Address Resolution Protocol (ARP)-based detection [7] is a notable method example used to identify scanning activities on an internal network. This method uses the characteristics of the ARP protocol to discover irregularities in network traffic patterns that are created as a result of malicious scanning behavior. A scanning host will raise alarms only when doing ARP activity that is beyond the standard normal one which, in turn, is associated with a device scanning within the network. In most cases, ARP is necessary to map IP addresses to MAC addresses, thus, establishing a communication process among devices residing on the same local network. In contrast, a scanning host will create a burst of ARP requests as it focuses on getting the list of active devices. As the requests are sent out over the entire local network, the level of ARP traffic increases noticeably.

To capture those scanning activities, the ARP-based detection systems are going to evaluate the frequency, volume, and the pattern of ARP requests. By doing so they set the basic behavior of ARP standards, besides taking into account such criteria as the average number of requests per minute and the type of communication between the devices.

In contrast, **external detection** is the ability to spot external hazards outside the network. Typically, these hazards are cybercriminals, state-sponsored actors, or automated bots trying to break into the network. By and large, the primary focus is on examining and tracking incoming traffic effectively, thereby eliminating potential threats before they can breach the network's defenses.

Therefore, firewalls and network-based IDS/IPS (NIDS/NIPS) are significant for external detection as they form the main part of the whole external detection itself.

DNS-based detection is among one of the best ways to discover external threats. Security systems can recognize unlawful activities by analyzing DNS queries and responses, such as domain generation algorithms (DGAs) [8] used by malware to communicate with command-and-control (C2) servers, or phishing attempts using deceptive domain names. DNS-based detection is an alert system to potential attacks in their early stages and it is aimed at helping security teams to take preventive measures. The following chapter will provide more insight into this type of detection, while our approach will be carefully looked at in detail for its efficacy and practical uses.

3.3 Firewalls

Firewalls are among the most fundamental and critical parts of a network's defense. The key activity that firewalls perform is to limit traffic entering and leaving a network, based on a set of security rules. This way, they help in preventing unauthorized access to a private network or from a private network, creating the barrier between the internal trusted network and the untrusted external networks, like the internet. This, in-turn, guarantees the safety of a network from data breaches and other forms of attacks.

Firewalls have greatly developed in current times, their function has been widened so that they can now encompass stateful inspection, deep packet inspection, intrusion prevention, and application-level filtering.

Packet-filtering firewalls are among the oldest and most elementary types of firewalls, that have become essential for the security of the network. Such firewalls operate at two layers of OSI Protocol Stack - the network layer (Layer 3) and the transport layer (Layer 4). They do this by analyzing the packet's source and destination IP addresses, port numbers, and protocol types. Only if a packet corresponds with an existing rule, it will be let through; otherwise, it will be disregarded (see Figure 17). The simplicity and effectiveness of packet-filtering firewalls make them a straightforward tool for monitoring network traffic and providing protection against basic threats such as unauthorized access attempts, port scans and denial-of-service (DoS) attempts.

These firewalls are designed to quickly evaluate packets, determining whether to allow or block them, based on the rules that have been previously defined, leading to the least possible time of latency in network communication.

However, while packet-filtering firewalls have their attractive characteristics, there are quite a few downsides: these firewalls fail to capture the data packet payload, which puts them at risk for sophisticated attacks that exploit application-specific vulnerabilities. They can not discover malware or even other harmful content hidden within the seemingly legal web traffic. Moreover, packet-filtering firewalls are unable to interpret the nature of communication. This limitation means they might block legitimate traffic that deviates from predefined rules or, conversely, allow harmful packets to pass through if they mimic authorized communication.

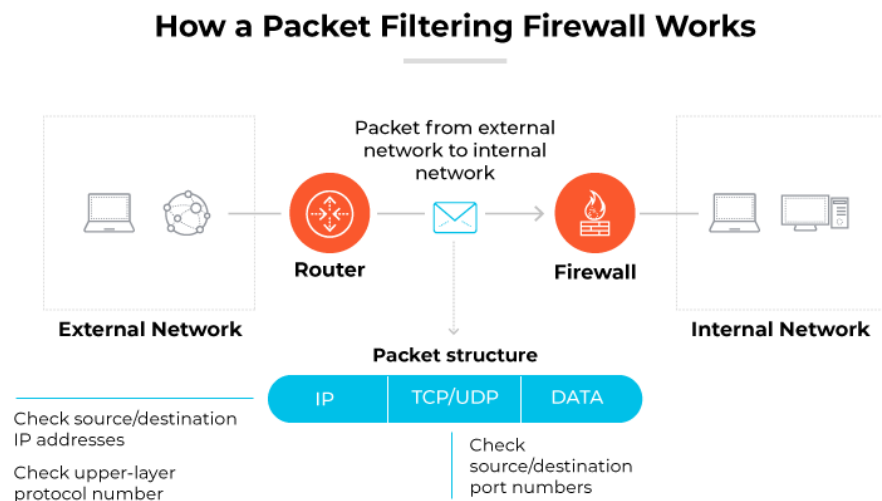


Figure 17: Packet Filtering Firewalls. Source: paloaltonetworks.com

Stateful inspection firewalls, or dynamic packet-filtering firewalls, are the kind of digital network security technology that is far more modern in terms of development, if to be compared with packet-filtering firewalls. Unlike their simpler counterparts, which deal solely with individual packets alone, stateful inspection firewalls create dynamic records of various network connections, essentially "remembering" the current states of these ongoing connections.

Their main layers of operation take place at the network layer (Layer 3) and the transport layer (Layer 4) of the OSI model. They maintain a state table (or state database) that shows the details of the active connections, such as the source and destination IP addresses, port numbers, and sequence numbers (see Figure 18).

Stateful inspection firewalls have a number of advantages over basic packet filtering firewalls: they

track the state of connections in order to allow a deeper analysis of network traffic, which means that it becomes difficult for attackers to bypass security with methods such as spoofing of IPs or port scanning. Hence, these firewalls block and identify highly advanced attacks that manage to exploit the flimsiness of stateless firewalls. For instance, they can notice and stop TCP session hijacking, where an attacker attempts to take over an existing, legitimate connection.

Despite their advanced security capabilities, stateful inspection firewalls have to deal with several disadvantages. One of the real challenges is that the maintenance of the state table requires a lot of processing power and memory, which can lead to performance problems and scalability issues. These firewall devices are really difficult to set up and manage, requiring specialized expertise, and increasing administrative overhead. There is also a risk of the electronics causing the jamming of the network since the applications have complex or dynamic communication patterns. The operations that are required might cause latencies, which, in turn, affect the proper functioning of the real-time applications.

This technology has another weakness, which is the state table exhaustion. With the connection that is either open or timed-out, the attacker will manage to take the full resources of the firewall away. In conclusion, it is obvious that the initial cost as well as the cost of maintenance are higher compared to stateless firewalls, thus making them a more expensive option.

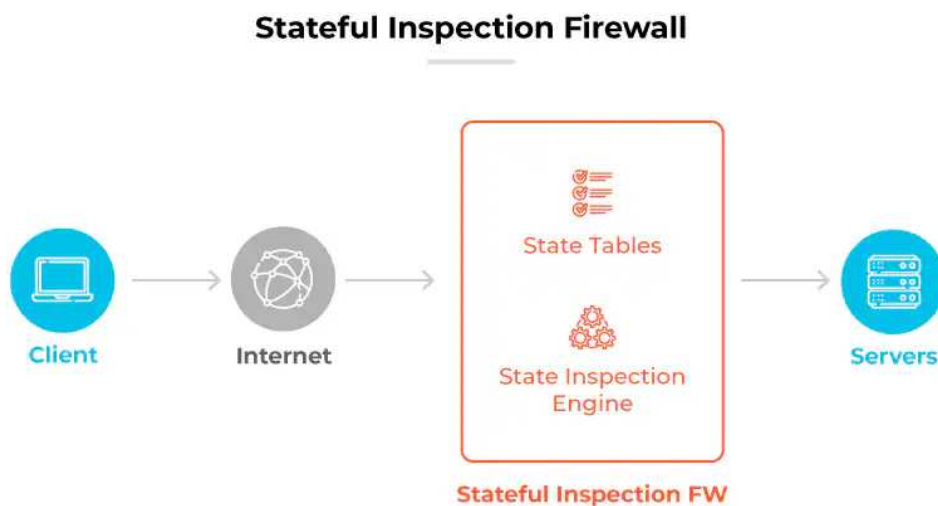


Figure 18: Stateful Inspection Firewalls. Source: paloaltonetworks.com

Proxy firewalls, which are also called application-level gateways, are another kind of security software that is designed to be an intermediary between the end user and the resources they connect to, with an aim at the safety of the network on a higher level. Unlike traditional firewalls that simply filter packets, proxy firewalls take a more active role in communication. They work on a higher level than the other two layers of the OSI model (Layer 7), which makes it possible for them to study the network traffic on a much wider scale and thus protect the system from a bigger range of threats. Proxy firewalls monitor all client requests that are to go to external servers and execute such requests on behalf of the clients.

The external servers' answers in turn are sent to the proxy firewall that sends them to the clients straight from it (see Figure 19).

Hence, when a user requests a web page, the request goes to the proxy firewall first. The firewall examines the content of the request at the application level, checking for malicious code, inappropriate content, or violations of security policies. Once the request is inspected, the proxy firewall forwards it to the destination server using its own IP address. This implies that the internal user's real IP address is hidden, which means extra security and privacy are placed.

However, the in-depth inspection can lead to performance degradation, impacting network speed, and if a proxy firewall is compromised, it could become a single point of failure, creating a security vulnerability.

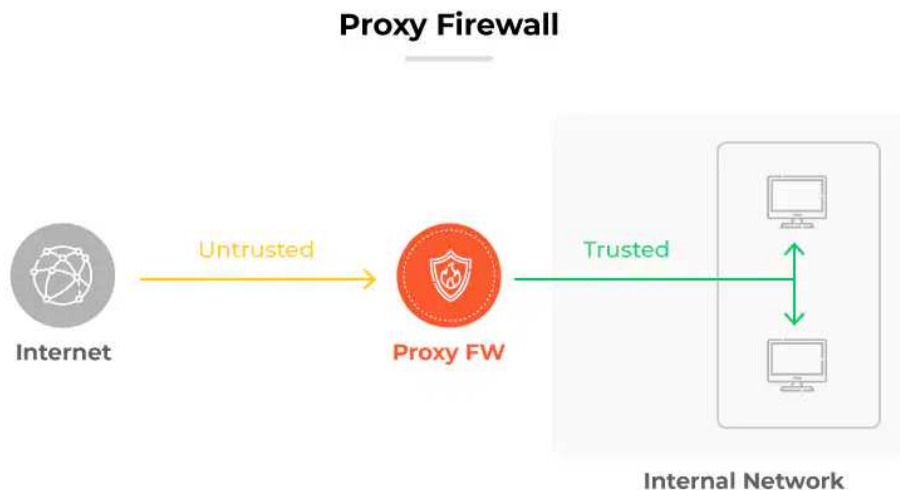


Figure 19: Proxy Firewalls. Source: paloaltonetworks.com

Finally, **Next-Generation Firewalls (NGFWs)**, far more sophisticated than their predecessors, are the fulcrum of security technology surpassing the grindstones that traditional ones are. This is possible by consolidating a wide range of advanced features into a single, unified system. This is a tool that can prevent against a variety of different cyber hazards by producing a multilayered defensive structure that includes combining deep packet inspection, application-level security, intrusion prevention, and real-time threat intelligence (see Figure 20).

Next Generation Firewalls provide a lot of key features:

- Deep Packet Inspection (DPI): with the use of deep packet inspection, the NGFWs are akin to digital X-ray machines which are capable of not only scanning the headers but also the contents of the data packets. Thus, these instruments can be used for revealing the obfuscated threats such as malware, harmful scripts, and exploits that might be overlooked by traditional firewalls.
- Application-Level Security: the NGFW capabilities, which are dedicated to the application layer, can process the protocols and handle the behavior of the different applications, such as web browsing and email.
- Intrusion Detection and Prevention Systems (IDS/IPS): Next-generation Firewalls come outfitted with their in-built IDS/IPS capabilities that are capable of tracking the communication on the network to patterns (if any) that may indicate attacks. Therefore, such systems can raise an alert when and if there is a problem and may also block the traffic feeds with the dangerous files.

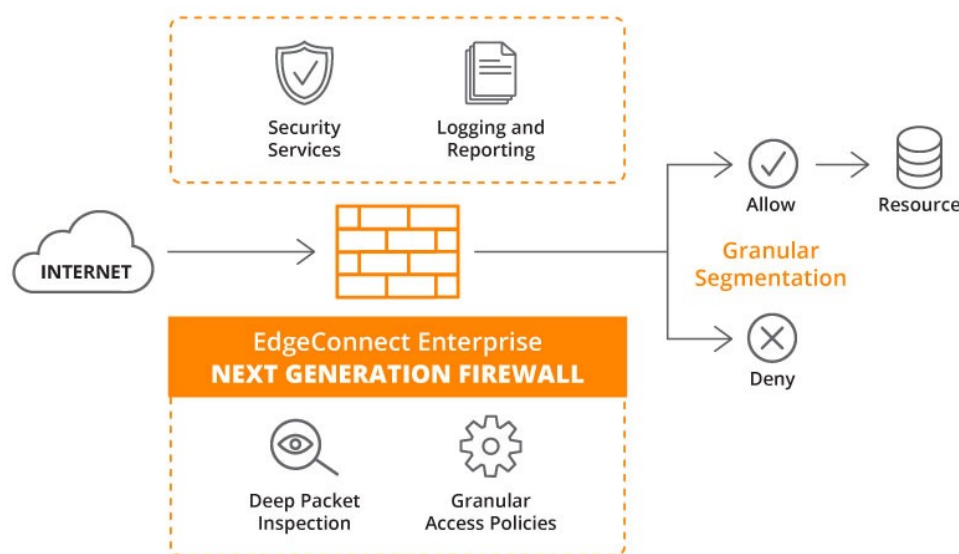


Figure 20: Next-Generation Firewalls. Source: arubanetworks.com

3.4 IDS and IPS

After exploring the importance of firewalls and the various types available, it is essential to delve deeper into network security by examining other crucial tools: Intrusion Detection Systems (IDS) and Intrusion Prevention Systems (IPS). In the cybersecurity world, these tools are vital in maintaining the safety and security of network traffic. Despite the fact that these security systems have few common features, they possess different functionalities that drive network security to a multi-layered defense system against a substantial number of cyber threats.

Intrusion Detection Systems (IDS) are considered passive security mechanisms used to watch and monitor network traffic for any doubtful activities and potential threats. They can detect and alert administrators about these threats, allowing for a quick response.

IDS can be categorized in two ways, mainly Host-Based IDS (HIDS) and Network-Based IDS (NIDS).

Host-Based IDS (HIDS) monitor activities on individual hosts or devices, such as file changes, system logs, and application activities. While, **Network-Based IDS (NIDS)** check the whole network segments and they do the monitoring of traffic passing through network interfaces. This approach is beneficial for detecting widespread network anomalies.

On the other hand, an **Intrusion Prevention System (IPS)** refers to a network security device that inspects network traffic for the purpose of detecting and preventing given threats, but, unlike Intrusion Detection Systems, which only signal administrators about possible security breaches, IPS manages to go one step ahead by automatically and simultaneously carrying out actions to hinder or lessen those threats. This proactive method ensures that threats are thwarted in real-time, thus reducing the likely damage to the network.

IPS can also be categorized in the two same ways as IDS: Host-Based IPS (HIPS) and Network-Based IPS (NIPS).

A well-planned and optimized network architecture is crucial for facilitating the detection of potential threats. The functional of IDS and IPS is a matter of installation points, and the selection of these sections must be made rationally and after taking into account traffic flow and the position of assets. Network segmentation is a kind of technology that is used to divide a large network into many different, isolated segments or zones. Each of these segments has its own security policies and controls, which, in turn, limits threats from spreading and makes more regulation of security management possible. By the incorporation of network partition, the going concern is that cybersecurity software will be able to create safer zones that would protect vital data from infiltration and, this way, minimizing the risk that all data will be compromised in case of a breach. Moreover, it permits the targeted placement of IDS and IPS for more efficient

monitoring and protection (see Figure 21).

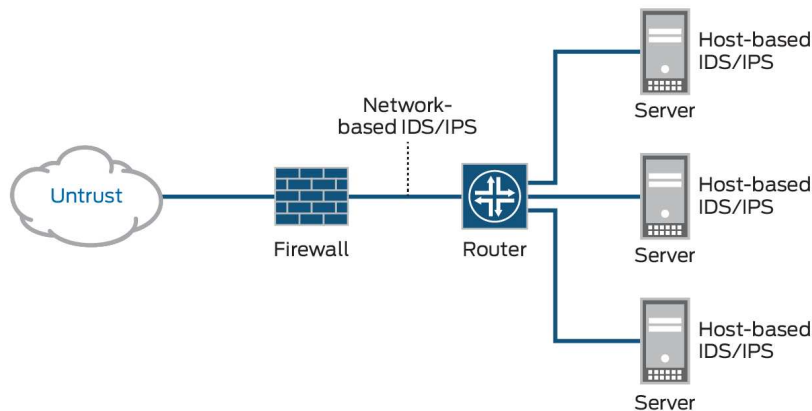


Figure 21: Position of IDS and IPS in an Enterprise Network Example. Source: juniper.net

In fact, when these systems are intelligently placed at some critical nodes of a network, they can eavesdrop on packets or inspect and analyze the data packets or packet flows (a flow is a combination of packets that pass through the control point or node).

The points where IDS and IPS are set up could be:

- Network Perimeter: IDS/IPS can be deployed to the network's edge as the first shield against outside threats. Since the spot they are located at is such a strategic one, the capability to identify any malevolent activities attempting to come into the network is present from the start. Perimeter IDS/IPS provide a view of all internet traffic, ensuring side surveillance of the network. The incorporation of these tools with the perimeter firewalls not only provides a number of layers of defense but also facilitates overall security. However, the network perimeter is a high-traffic area, which can lead to performance bottlenecks and necessitate powerful hardware to manage the load. Moreover, the diverse nature of legitimate external traffic can lead to a higher probability of false positives, consequently causing security teams to be flooded with alerts. At the same time, perimeter IDS/IPS may not have enough information on what takes place inside a network, and, as a result, it is a difficult task to associate an event with the whole network.
- Inter-Zone Firewalls: installing IDS/IPS in this position of the network gives the ability to detect push-through behavior transversally. They fulfill the function of catching suspicious events that shift from side A to side B. Therefore, inter-zone placement ensures that traffic adheres to established security rules, maintaining segment integrity. IDS/IPS can perform deep packet inspection, identifying complex attack patterns that might not be visible at

the perimeter. However, the additional processing required for deep packet inspection may introduce latency and performance issues. At the same time, this strategy only watches traffic among segments, fully missing dangers that are tightly kept in a single segment. Automatically generated false positive results setting off the data-catching mechanism is a recurring condition, with the multifaceted warnings of the different patterns needing detailed setting and management.

- Demilitarized Zone (DMZ): placing IDS/IPS in the DMZ allows for monitoring traffic that will be talking with both external and internal networks. This isolated zone makes it easier for the network to detect new threats quickly if they come through the DMZ. IDS/IPS in the DMZ can defend public servers like the web server and mail server from external attacks. Checking the traffic in the DMZ can instantly catch infected systems, hence stopping them from going further inside little by little. Nevertheless, like other placements, DMZ IDS/IPS may introduce latency due to deep packet inspection and real-time analysis, necessitating careful management to balance performance and security.

3.5 Network Traffic Analysis

Network Traffic Analysis (NTA) is an important matter regarding security, which is the study of network data in full extent to identify patterns, anomalies, and potential security threats. The scrutiny of data packets transmitted across a network enables NTA to gather information about network behavior, which is very useful for organizations to detect and react to unethical activities, performance issues, and policy violations. This method of analysis, crucial for maintaining the integrity, availability, and confidentiality (CIA) of network activity, is indispensable for ensuring the security of the enterprise.

The main components of NTA are, first of all, the uninterrupted monitoring of the network's traffic, which is done by collecting the packets of data while they are being sent from one device to another. Subsequently, this data will be thoroughly processed by identifying the source, destination, type, and amount of traffic. NTA can monitor unusual behaviors by analyzing these characteristics, signaling a security threat that needs to be looked at.

Flow data is one of the main methods that NTA employs for collecting and evaluating traffic data flow. Specifically, flow data gives a report of the traffic between two endpoints, and it includes the IP address details, port numbers, and protocols used. For example, a rocketing traffic from one single IP address might be a sign of a DDoS attack, whereas an unusual port activity can be a sign of a network vulnerability exploitation.

NTA also performs deep packet inspection (DPI) to look at the details of the packets to a greater

extent. DPI means evaluating not only the header and metadata but also the full payload of each packet, helping to do a thorough check. This depth of scanning allows the identification of suspicious content, like known malware signatures intentionally or unintentionally downloading a malicious program, suspicious file transfers, or the presence of botnets using command-and-control communication. DPI integration with machine learning algorithms ensures setting up NTA systems for taking actions to the upcoming threats and hence, detecting them better over time.

However, the introduction of NTA is not seamless. The multitude of data streams that constantly flow through big companies' networks can cause the collection and analysis of data to become resource-heavy. High-speed networks contribute data volumes that are really huge, and these data should be processed on time so that the necessary information is provided. This, therefore, needs the required infrastructure and, specifically, big data tools that can work with the data. Privacy issues are also a big deal with NTA, especially with deep packet inspection. Analysis of the information transmitted in network packets can result in the accidental exposure of the personal data, making it an issue that has ethical and legal considerations. Organizations also need to align safety concerns with user privacy in ways such as using methods like filtering or scrambling the data.

The benefits of NTA, in spite of the problems of its implementation and privacy issues, make it a necessary part of any cybersecurity plan as it ensures network security and functionality.

3.6 Classification of Detection Algorithms

IDS and IPS can implement detection methods that are mainly categorized into four types: specification-based detection, misuse-based detection, anomaly-based detection and behavior-based detection.

Specification-based detection, also known as stateful protocol analysis (SPA), is a way of identifying a method that is compared to the behavior of the network with the previously known rule or specification that describes how the system should behave. This method consists of creating a model or a profile in which the expected behavior of the network applications and protocols is represented. If the network behaves differently from the specifications, the alert is displayed. This technique requires inserting a definition of the normal operation and a description of which parameters are equipped or are used by the network's applications and services.

For instance, systems like STAT (State Transition Analysis Technique) [9] utilize state transitions to represent legitimate behavior. Any deviation from these defined states is flagged as a potential intrusion, making it effective for detecting unauthorized activities within predefined protocols.

Additionally, frameworks like NetSTAT (Network State Transition Analysis Technique) [10] extend the capabilities of state-based detection to network environments, focusing on specifying and detecting unauthorized state transitions in network protocols.

Hence, security experts write precise specifications that clearly identify the types of activities and interconnections allowed within the network. The specifications thus made are employed in the network traffic for the monitoring purpose. This type of detection is very accurate and, therefore, discovers deviations from normal behavior, making it a powerful tool for capturing unknown or zero-day attacks.

Consider the Network Specification Language (NSL) [11], which allows the description of network behavior and policies in a structured language. Activities not conforming to these specifications are identified as suspicious.

Another example is the P-BEST (Policy-Based Event Selection and Tracking) method [12], which uses predefined policies to monitor events and track activities, identifying any deviations from the established policies.

Misuse detection, also called signature-based detection, is a method used in IDS and IPS to detect cyber threats that have known patterns of malicious activity. Misuse detection is performed by the comparison of current different system and network activities with the known database of attack signatures. These signatures are pre-defined patterns that determine exploits, such as specific sequences of network packets, system calls, or particular fragments of the

byte in malicious payloads. A perfectly designed database with the attack signature of every known attack is the main point of misuse detection. These signatures are created and improved by security experts and vendors who keep track of the latest infections. In other words, each signature acts as a unique fingerprint for an attack and gives out the specific features that differ from benign activities.

The detection system persistently scrutinizes network traffic, system logs, and other such data to keep an eye on possible threats. To do so, the pattern-matching algorithms compare incoming data with the signatures that have been stored in the database. At the same time, the system marks the suspicious activity so it can be examined further. After identifying the suspicious activity, the IDS produces an alert to let the security personnel know. The signature database is subject to only intermittent changes, so to stay operable it must be constantly updated. In fact, security teams and vendors must keep checking for and analyzing new threats. This process of continuous updates may use up a lot of resources and time. But the biggest weakness of misuse detection is its reliance on known signatures. It does not have the ability to detect zero-day attacks or other new threats not covered in the current list of signatures.

Criteria for determining misuse is the use of systems like Snort [13] or Suricata [14], which are the commonly used IDS/IPS solutions. Snort employs a rule-based language to define traffic patterns and uses these rules to detect potential threats in real-time, such as Nmap scanning techniques (SYN scans, ACK scans, FIN scans, and others). For example, a SYN scan involves sending TCP packets with the SYN flag set without completing the three-way handshake. Snort has rules that look for a high rate of SYN packets without corresponding ACK responses, which is indicative of a SYN scan. Similarly, rules can be created to detect patterns associated with other types of Nmap scans, such as FIN scans (sending TCP packets with only the FIN flag set) and XMAS scans (sending packets with multiple flags set).

Suricata, on the other hand, supports multi-threading, which allows it to process more data in parallel compared to Snort, making it well-suited for high-speed networks, and it also uses rule sets similar to Snort's to identify suspicious activity.

Systems like Bro (now Zeek) [15] also incorporate signature-based detection along with other approaches to offer a more solid security solution and apply the built-in configuration scripts to recognize and log suspicious activities.

Anomaly detection is another approach to identifying and avoiding threats that are likely to be omitted by the traditional security tools; thus, it is one of the most important methods of security. Besides establishing itself as a digital watchdog, these systems continuously monitor the network and system behavior for discrepancies from the norm, identifying the hidden threats

that could otherwise be left ignored by the traditional protection measures. To outperform the network invasions of any kind, a baseline has to be created that describes the normal activity of its systems on the network. This baseline is an important point of reference for the identification of deviations that can be indicators of likely threats. To conduct this exercise, the system keeps track of a variety of network metrics and performance indicators over time, like the volume of data that is being moved as produced by the network (unusually high data transmission volumes could indicate a Distributed Denial of Service attack), characteristics of the data packets, the traffic protocols used by communication, and patterns of CPU, memory, and disk access (unexpectedly high CPU, memory, or disk usage might signal malware or a compromised system, or it could also reflect unauthorized access or system manipulation).

Through the extensive dataset collected, then the system will be taught of the behaviors and actions that are categorically the usual conditions. On a daily basis, contemporary anomaly detection systems largely rely on the capability to adjust to new conditions by means of machine learning algorithms that go hand in hand with evolving network usage patterns and developing system performances. As a result of this progressive learning potential of the system, the baseline does not deviate from accuracy and purpose even when the network environment changes. When an abnormality is detected, the system rates the severity and background to ascertain the deviation. Accordingly, the system may generate notifications to the security teams to be able to quickly look into the issue and respond. This tried and true way of being proactive is what makes it possible to spot the threats and prevent them from becoming significant security incidents. The accuracy of setting and the right level of sensitivity are practically a key factor to be taken into account. A system that is overly sensitive might lead to a significant increase in the number of false alarms, thereby overwhelming the security department with alerts for non-threatening anomalies. In contrast, a system that does not have increased sensitivity may let serious problems go by unnoticed. Moreover, all anomalies do not imply malice: some deviations may be caused by legitimate but unusual network traffic or system events.

Various algorithms are put into work in anomaly-based detection to identify deviations from the norm:

- K-Means Clustering [16]: this is the algorithm that is used to divide the data into K clusters. Each cluster represents a pattern of normal behavior. Data points that do not fit into any cluster are considered anomalies. As an illustration, network traffic data can be segmented using attributes like packet size and frequency, with outliers being highlighted as potential threats.
- Principal Component Analysis (PCA) [17]: through PCA, one can select only the most important feature from the data, thus, decreasing the dimensionality while retaining almost

all the original information of variability. This technology assists in the visualization of the patterns in high-dimensional data, thus making it more convenient to detect the anomalies. In the field of cybersecurity, PCA can be used to recognize the irregular patterns in the network traffic by analyzing the core components of the data.

- Gaussian Mixture Models (GMM) [18]: GMMs assume that data is a combination of several Gaussian distributions. Statistical methods are used to estimate the parameters of these distributions, allowing the detection of anomalies as data points that do not fit well within any of the distributions. This method is effective in identifying advanced threats in network traffic.
- Support Vector Machine (SVM) [19]: SVM is a supervised learning algorithm that can be used for anomaly detection by first training the model on normal behavior. The SVM then labels new data points based on their proximity to the normal behavior boundary, marking as anomalies the outliers. SVM is particularly great when it comes to identifying deviations in system resource usage.
- Isolation Forest [20]: this algorithm isolates observations by randomly selecting a feature and then randomly selecting a split value between the maximum and minimum values of the selected feature. The anomalies are those points that require fewer splits to isolate. Isolation Forest is effective in detecting anomalies in high-dimensional data sets.
- Neural Networks and Autoencoders [21]: these deep learning techniques can learn complex patterns of normal behavior by training on large datasets. Autoencoders, for instance, compress the input data and then reconstruct it. The reconstruction error is used to detect anomalies, with high errors indicating significant deviations from the norm.

Moving forward, **behavior-based detection** is a dynamic detection approach that focuses on identifying threats by monitoring and analyzing the behavior of users, applications, and network traffic. The foregoing creates models of normal behavior and flags deviations as potential threats, thus making it such a powerful tool for detecting both known and unknown attacks.

Indeed, behavior-based detection is constituted on the postulate that evil activities often demonstrate a pattern of behavior that is different from the regular activities. These systems, in particular, do so by keeping a constant eye on users' behaviors, thus paving the way for proper, invisible detection of these types of agents. The baseline is expressed through numerous factors, user actions, application processes, and traffic patterns along the network, among them.

One of the primary strengths of behavior-based detection is its ability to identify zero-day attacks and evolving threats. This method is not restricted to the use of predefined signatures or rules (as signature-based and specification-based detection), so it can detect previous attacks by

the deviation from normal behavior.

In addition to the benefits of behavior-based detection, some troubles are also set off. The most significant one is the potential for false positives. Since this mechanism is founded on the idea of marking all the displacement from the reference point as a potential threat, real activities that are a bit unusual can also be declared as dangerous activities causing alerts. This calls for the security teams to engage in a detailed analysis of the information and interpret the alerts so as not to disrupt the smooth operation of the system.

In contrast to anomaly-based detection, which also identifies deviations from normal behavior as previously mentioned, behavior-based detection emphasizes a more holistic and adaptive learning process. These two approaches work by detecting abnormal activities based on a comparison of the existing activity with a baseline of typical behavior patterns. On the one hand, behavior-driven detection is gaining attention by showing that machine learning and human-computer interaction are very different when it comes to detecting and exposing potential threats, whereas the second method is based on tracking the statistical deviations from the norm.

Several algorithms and techniques are utilized in behavior-based detection to accurately identify anomalies:

- Hidden Markov Models (HMMs) [22]: they represent systems with latent (hidden) states. In the situation of behavior-based detection, HMMs can represent a set of users' movement sequences or network events. By processing the likelihood of different sequences, HMMs can find out abnormal patterns that may suggest potentially hazardous behavior.
- Long Short-Term Memory Networks (LSTMs) [23]: LSTMs are of the recurrent neural network (RNN) type, which can learn and remember over long sequences of data. They are mainly efficient in time-series data anomaly detection, for instance, user login patterns or network traffic flows. LSTMs are capable of observing minor divergences from normative behavior that are incomprehensible to other approaches.
- Bayesian Networks [24]: these probabilistic graphical models represent a set of variables and their conditional dependencies via a directed acyclic graph. In behavior-based detection, Bayesian networks can be used to model the relationships between different activities and identify combinations of events that are likely to indicate a threat.
- Random Forests [25]: these ensemble learning methods combine multiple decision trees to improve classification accuracy. When it comes to behavior-based detection, random forests are able to process various features of user or network behavior to detect anomalies. The method's robustness to overfitting and ability to handle large datasets make it a powerful tool for identifying complex threats.

Chapter 4

DNS-based Detection of Network Scanning

4.1 History of DNS

Earlier than the DNS was brought in, the leading networks, such as ARPANET [26] (the precursor to the modern internet) and the internet, began to use a central HOSTS.TXT file (that was distributed manually) to link domain names and IP addresses, administered by the Stanford Research Institute. This approach soon turned out to be unrealistic and difficult to use because of the necessity of real-time link updates. In fact, in the early 1980s, Paul Mockapetris was given the challenge of creating a scalable and distributed system to convert human-readable domain names into IP addresses. It is necessary to convert them because, while domain names are easy for humans to remember, computers and networking hardware use IP addresses to communicate. To fix the issue, Mockapetris formed the first DNS standards, which were reported in RFC 882 and RFC 883 and afterward refined to RFC 1034 and RFC 1035. These were the documents that established the basis of the Domain Name System (DNS), and they were the ones through which its architecture and operational principles were presented.

In addition, he introduced the main concepts of the system, which include the hierarchical domain name system as well as a more efficient and scalable domain name system. This approach allowed for the efficient and scalable management of domain names and their corresponding IP addresses. His design consists of such things as support for recursive and iterative queries, caching mechanisms to improve efficiency, and extensibility that allows infrastructure upgrades and changes in the internet's structure.

In 1983, the Domain Name System (DNS) was finally born. This distributed system offered scalability, fault tolerance, and reduced load compared to the central file.

At present, DNS still performs an indispensable role in the proper working of the Internet, and it is being updated to the new requirements of the future.

4.2 DNS Basics

Moving into the technical aspects, the domain name space is structured hierarchically, resembling a tree with distinct levels (similarly to the structure of the Unix filesystem) that organize internet addresses based on their purpose and ownership. At the apex of this hierarchy is the

Root Level, denoted by a single dot ("."). The Root Level, managed by the Internet Corporation for Assigned Names and Numbers (ICANN) [27], coordinates the distribution of queries by providing pointers to the appropriate **Top-Level Domain (TLD) servers** across the globe. There are currently 13 root nameservers [28], distributed globally, that form the foundation of the DNS system. These nameservers do not contain the IP addresses themselves, but they act like a directory, pointing recursive resolvers in the right direction based on the TLD in the domain name being queried.

Directly below the Root Level in the DNS hierarchy are the Top-Level Domains (TLDs). These are divided into two main types: **Generic Top-Level Domains (gTLDs)** such as .com, .org, and .net, which are globally recognized and used across various sectors, and **Country Code Top-Level Domains (ccTLDs)** like .uk, .de, and .jp, which are specific to countries and territories.

Each TLD is managed by designated registries responsible for overseeing domain registrations and the delegation of second-level domains. Second-Level Domains are positioned directly beneath their respective TLDs and typically serve as unique identifiers registered by organizations or individuals. For instance, in "example.com," "example" is the second-level domain.

Subdomains are extensions of second-level domains, further dividing internet resources to facilitate organizational structure or service differentiation. Examples include "mail.example.com" or "blog.example.com," each operating under the authority of its parent domain (see Figure 22).

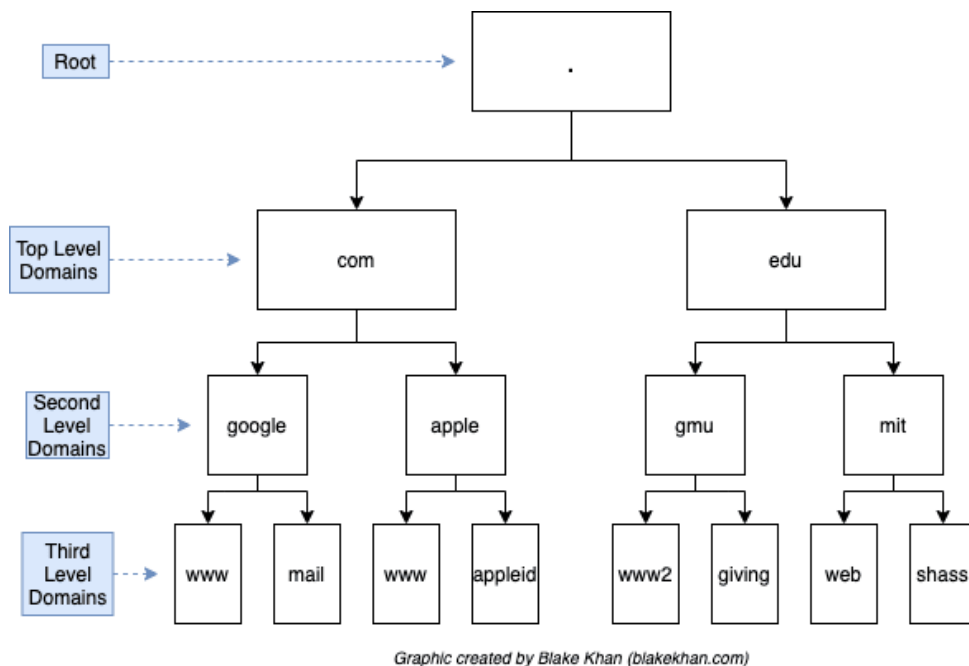


Figure 22: Domain Name Hierarchy.

As we know, DNS is designed with decentralized administration and a distributed database to accommodate the diverse authorities involved in managing internet domain names.

Hence, **DNS zones** are pivotal to the hierarchical structure of DNS, dividing the domain name space into manageable segments. Each zone is delegated to a specific administrative entity, allowing different organizations or individuals to independently manage their own DNS records within their designated zone.

The primary types of DNS zones include:

- **Primary Zone:** it is the authoritative source of DNS information for a domain. DNS records (like A records for IP addresses) within a primary zone are stored locally on authoritative nameservers responsible for that zone. Changes and updates to DNS records are made directly on these primary nameservers.
- **Secondary Zone:** it is a read-only copy of a primary zone, and it is synchronized periodically with the primary zone, ensuring consistency of DNS records. Secondary zones provide redundancy and fault tolerance, enhancing the availability of DNS resolution services.
- **Stub Zone:** it contains minimal DNS records, primarily NS (Name Server) records, and it can be used to forward queries to the authoritative nameservers.

Each DNS zone comprises several essential components and characteristics:

- **Zone File:** it is a text file that stores DNS records for a specific zone. It contains resource records (RRs) such as A records (IPv4 addresses), AAAA records (IPv6 addresses), MX records (mail exchange servers), CNAME records (canonical names), and others (see Figure 23).
- **SOA Record (Start of Authority):** this record is the first entry in a zone file and specifies authoritative information about the zone. It includes details such as the primary nameserver for the zone, the email address of the zone administrator, and various timing parameters (e.g., refresh, retry, expire, minimum TTL).
- **NS Records (Name Server):** they identify the authoritative nameservers for the zone and they delegate authority for the zone to these nameservers, directing DNS queries to the correct sources of DNS information.

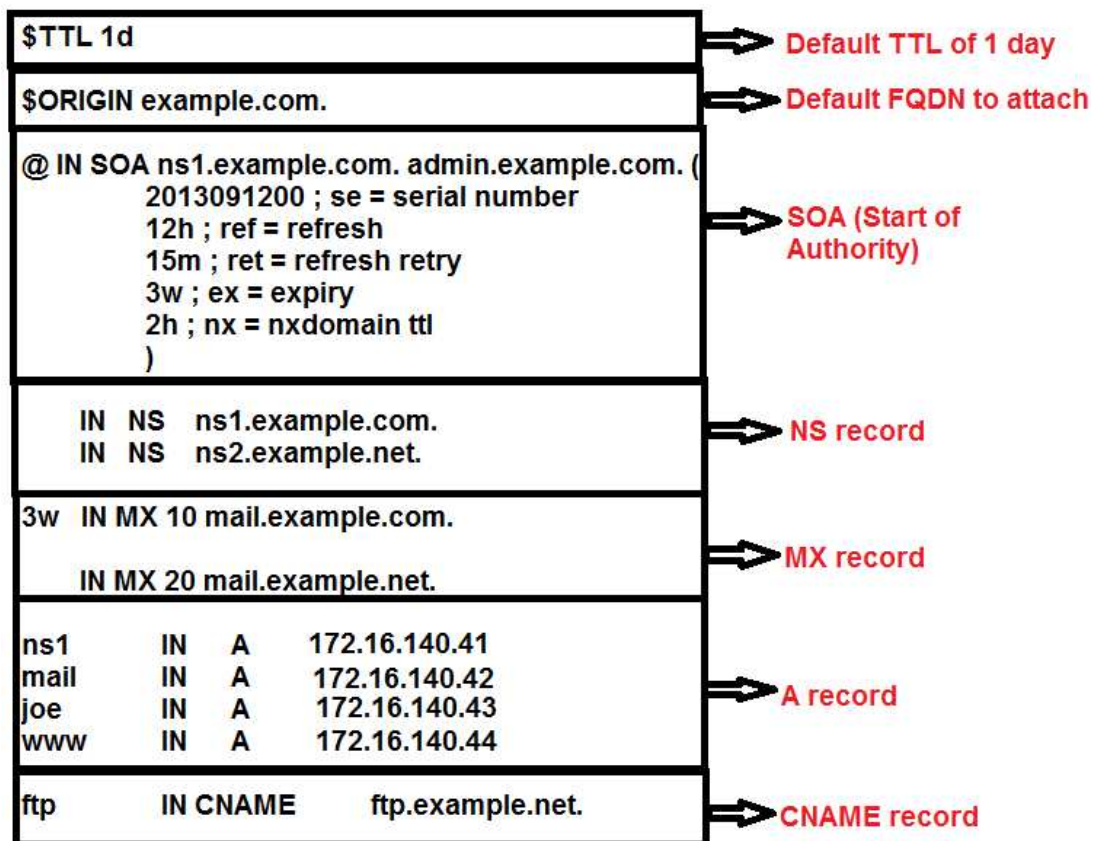


Figure 23: DNS Zone File Example.

After examining the DNS zones, it is crucial to delve into the detailed roles of name servers and resolvers within the domain name system.

Name servers play a fundamental role in the DNS system: they are responsible for managing and distributing domain name information within specific zones.

There are two main types of nameservers:

- **Root Nameservers:** these servers are essential to the DNS system because they are responsible for directing queries to the appropriate top-level domain (TLD) servers. As previously mentioned, there are 13 root servers, labeled A through M, placed around the world to ensure reliability and redundancy. When a DNS resolver initiates a query, it often begins by contacting a root nameserver and this initial step, as we will see in the next paragraph, is crucial because it helps navigate the vast expanse of the internet's domain name space efficiently.
- **Authoritative Nameservers:** they provide the definitive answers for specific domains within their designated zones. In fact, each DNS zone may have one or more authoritative name servers designated to hold and provide official DNS information for that particular domain or subdomain. When queried, these authoritative name servers respond with

precise and up-to-date DNS details, such as IP addresses associated with the domain's services, like websites or email servers. Therefore, unlike root name servers, which serve as navigational aids, authoritative name servers are the ultimate sources of truth for DNS data as they hold the actual DNS records, such as A records for IP addresses, MX records for mail servers, and CNAME records for aliases.

4.3 DNS Query Resolution

After introducing the fundamental principles of DNS and the various actors involved, we can now delve into the essence of DNS: the actual process of DNS resolution. Therefore, in this paragraph we will explore the roles of resolvers, the resolution process, and how queries are handled and resolved to ensure users can access the correct IP addresses associated with domain names.

Firstly, **resolvers** play a central role in the DNS system, acting as intermediaries that handle the complex process of translating domain names into IP addresses. Their function is essential, as they query various DNS servers to obtain the information necessary to resolve a domain name to its corresponding IP address. There are different types of resolvers, each with a distinct purpose in the resolution process:

- **recursive resolvers:** these resolvers are tasked with managing the entire DNS resolution process on behalf of the client. In fact, when a client (such as a web browser) initiates a DNS query, it sends the request to a recursive resolver, which is responsible for querying multiple DNS servers to locate the correct IP address for the domain name, as we will explain shortly in the resolution process.
- **iterative resolvers:** (also known as non-recursive resolvers) they handle DNS queries differently. Instead of managing the entire resolution process, iterative resolvers respond to client queries by providing referrals to other DNS servers that may have the answer. For example, when a client queries an iterative resolver, the resolver might respond with the address of a TLD server rather than the final IP address. Then, the client must query the TLD server and possibly additional authoritative servers to obtain the final IP address. Obviously, this approach requires the client to handle more of the resolution process, but it can reduce the load on individual resolvers.
- **stub resolvers:** they are simple DNS clients typically embedded within operating systems or applications. Stub resolvers do not handle the entire resolution process themselves. Instead, they initiate DNS queries and forward them to a recursive resolver or local DNS server for resolution, thereby simplifying the client's role in the DNS resolution process.

When discussing the **DNS resolution process**, a client initiates a request for the IP address associated with a domain name. The process begins with the stub resolver, which forwards the query to a recursive resolver (typically managed by the Internet Service Provider or configured on the device). If the resolver does not already have the IP address in its cache, it starts the resolution process by querying the root name servers. These servers respond with a referral to a TLD (Top-Level Domain) server responsible for the top-level domain of the queried domain name. Based on the information received from the root name servers, the resolver then queries the appropriate TLD name servers. For instance, if the domain is "www.google.com", the resolver queries the ".com" TLD name servers (see Figure 24).

Subsequently, the recursive resolver queries the TLD server, which provides a referral to the authoritative name server for the specific domain. Finally, the recursive resolver communicates with the authoritative name server, which maintains the authoritative DNS records for the domain and provides the corresponding IP address. The resolver then returns this information to the requesting application or user.

How does DNS resolve IP

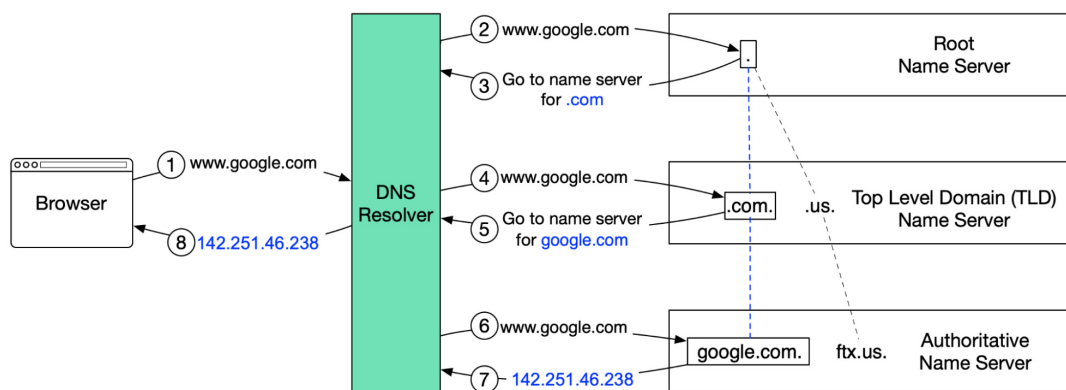


Figure 24: DNS Query Resolution Example. Source: bytebytego.com

4.4 Reverse DNS Query Resolution

Reverse DNS (rDNS) is a critical component of the Domain Name System that performs the reverse operation of the more commonly known forward DNS. In fact, instead of translating domain names into IP addresses, it translates IP addresses back into domain names.

In a reverse DNS lookup, the query begins with an IP address, and the process involves using a special domain called "in-addr.arpa" for IPv4 addresses and "ip6.arpa" for IPv6 addresses.

When a reverse DNS lookup is started, the resolver sends a query to the DNS server to find the PTR record corresponding to the given IP address. For example, the IP address 192.0.2.1 becomes 1.2.0.192.in-addr.arpa. The resolver then queries the root name servers, which respond with a referral to the appropriate top-level domain (TLD) servers for the "arpa" domain.

After the root name server gives a referral, the resolver queries the TLD servers that handle the "arpa" domain, and these servers provide details to direct the resolver to the authoritative DNS servers for the specific part of the IP address. The authoritative DNS servers are the final stop in this process, as they hold the PTR records that link IP addresses to domain names. When the resolver retrieves the PTR record, it sends this information back to the requesting client (see Figure 25). Reverse DNS is especially useful for email servers, as it helps prevent spam by checking the domain name for the email sender's IP address. Many email programs perform reverse DNS lookups as part of their spam filtering process, ensuring that the sending server's IP address matches the domain name in the email headers. It also helps outside of logging and networking functionality, enabling administrators to better identify and troubleshoot problems associated with specific IP addresses.

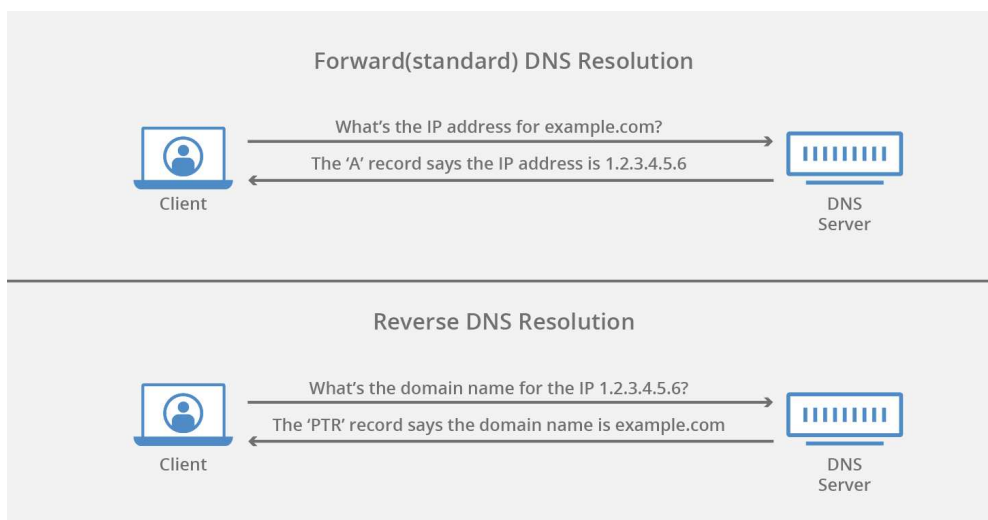


Figure 25: Forward DNS Resolution VS Reverse DNS Resolution. Source: dnswizard.com

4.5 DNS Caching

At this point, it is also essential to talk about caching methods within the DNS system, particularly the concept of Time to Live (TTL). In fact, caching plays an important role in the efficiency and performance of DNS, significantly reducing the load on DNS servers and speeding up the resolution process for end-users.

When you visit a web page, the computer's DNS resolver finds the IP address for the domain name and stores this information temporarily in a cache. This way, if you or anyone on the same network revisits the site, the resolver can quickly provide the IP address without repeating the entire lookup process. In this process, this storage can be controlled by the TTL value.

TTL is a boundary in the DNS records that describes to resolvers how long to store the details in their cache. The value is given in seconds and designates a DNS record's lifetime in the cache before it has to be reloaded each time.

Competent DNS configuration includes setting a TTL value for each record by authoritative servers, which ensures the balance of updated information and speedy performance.

A TTL starts its countdown cycle at the instant a DNS record is fetched for the first time. To illustrate, if a TTL is 3600 seconds (or 1 hour), the record is remaining in the cache for the next hour. All the requests that are being sent to the same domain name server during this period are answered from the cache, resulting in a decrease in the response time. When the TTL finishes, the record is deleted from the cache. If after this a new query for the domain comes up, the resolver now has to go through the resolution process again. One of the steps is to ask the root servers for the most recent information. This guarantees that each DNS record update is ultimately reflected across the internet after the TTL duration. Therefore, the main advantage of TTL in DNS caching is the improvement of efficiency by reducing the number of requests that have to be sent to authoritative servers, thus not asking them to do a lot of work, and hence, speeding DNS resolution for end-users.

Obviously, popular websites particularly benefit from DNS caching, making it faster for users who frequently visit them.

However, defining the appropriate TTL is the compromising point. A longer TTL means fewer queries to authoritative servers, but it also denotes changes to DNS records take longer to propagate. This can be an issue when quick updates are needed, such as changing IP addresses or adjusting load balancing.

When administrators establish TTL values, they prioritize the least-repetitive and most informative DNS records. Typically, longer TTLs are set for DNS records that do not change frequently, such as those pointing to stable IP addresses, with a minimum value often set at 24 hours in

these cases. This results in a decrease in accesses to authoritative servers and an increase in resolution speed for users. But differences in records come when TTL is the main leader. In fact, for records that change more often, like those used in dynamic environments or for load balancing, a shorter TTL (like 300 seconds) is better. This ensures changes propagate quickly, providing users with the most up-to-date information. In critical situations, such as during DNS changes or migrations, administrators might temporarily set the TTL to a very short duration (like 60 seconds).

Strategically setting TTL values is the key to DNS administrators finding the right mix between constancy and agility that will let the DNS services run smoothly, be highly productive, and react to policy changes. This contrast must be kept in mind in order to maintain seamless internet services and a way for everyone to get accurate information.

4.6 DNS Packet Structure

The DNS packet is an essential component of the DNS protocol and is a very necessary part of the communication between the clients and the servers. Having a deep understanding of its structure would simplify the process of DNS query and DNS response, how they are formed, and how they are processed. A DNS packet has a series of sections, and each one of them has a defined purpose.

The **header** part of the DNS packet is 12 bytes long, and it contains flags and codes that are letting know the type of query and response and the other attributes too (see Figure 26).

The header includes the following fields:

- Transaction ID: a 16-bit identifier assigned by the client and that matches responses to corresponding queries.
- Flags: a 16-bit field with various flags and options indicating query type (standard, inverse, or status), response status, and other control flags.
- Question Count (QDCOUNT): a 16-bit field specifying the number of questions in the Question section.
- Answer Count (ANCOUNT): a 16-bit field indicating the number of resource records in the Answer section.
- Authority Count (NSCOUNT): a 16-bit field showing the number of resource records in the Authority section.
- Additional Count (ARCOUNT): a 16-bit field specifying the number of resource records in the Additional section.

Whereas, here's a detailed breakdown of the header flags:

- QR (Query/Response): flag 0 for query and 1 for response.
- Opcode: flag 0 for standard query (QUERY), 1 for inverse query (IQUERY), 2 for server status request (STATUS), and 3-15 reserved for future use.
- AA (Authoritative Answer): it indicates if the responding DNS server is an authority for the domain name in the question section.
- TC (Truncated): it indicates that the message was truncated due to its length exceeding the maximum permissible size.
- RD (Recursion Desired): it is set by the client to request recursive resolution. If the server supports recursion, it will follow the chain of referrals to resolve the query.
- RA (Recursion Available): it is set by the server in the response to indicate that recursion is available.

- Z: it is reserved for future use (must be zero).
- AD (Authentic Data): it is set by the server to indicate that the data is authenticated by DNSSEC.
- CD (Checking Disabled): it is used by the client to disable DNSSEC validation.
- RCODE (Response Code): flag 0 for "no error", 1 for "format error" (unable to interpret the query), 2 for "server failure" (unable to process the query), 3 for "name error" (domain name does not exist), 4 for "not implemented" (query type not supported), 5 for "refused" (server refuses to perform the operation), and 6-15 reserved for future use.

DNS Header																																	
Offsets		0							1							2							3										
Octet	Bit	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
0	0	Transaction ID														Flags																	
																Q R	OPCODE			A A	T C	R D	R A	Z		RCODE							
4	32	Number of questions														Number of answers																	
8	64	Number of authority RRs														Number of additional RRs																	

Figure 26: DNS Header. Source: wikipedia.org

As we know, the DNS query aims to obtain details such as the corresponding IP address or other DNS records associated with the domain. The query message consists of a header and a question section (see Figure 27). The header includes the transaction ID and flags, while the **question section** contains:

- QNAME: the domain name for which the information is requested, fully qualified.
- QTYPE: the type of record being requested (e.g., A for IPv4 address, AAAA for IPv6 address, MX for mail exchange server).
- QCLASS: the class of the query, which is almost always IN (Internet).

Resource record (RR) fields

Field	Description	Length (octets)
NAME	Name of the requested resource	Variable
TYPE	Type of RR (A, AAAA, MX, TXT, etc.)	2
CLASS	Class code	2

Figure 27: DNS Request Resource Record. Source: wikipedia.org

On the other hand, the DNS answer is the response from the DNS server that contains the requested information (see Figure 28). The **answer section** of a DNS response contains:

- Name: the domain name to which the record applies.
- Type: the type of DNS record (e.g., A, AAAA, MX).
- Class: the class of the DNS record, typically IN for Internet.
- TTL (Time to Live): as explained before, it indicates how long the record can be cached before it should be discarded.
- RDLLENGTH: the length of the RDATA field.
- RDATA: the actual data of the record, such as an IP address.

Resource record (RR) fields		
Field	Description	Length (octets)
NAME	Name of the node to which this record pertains	Variable
TYPE	Type of RR in numeric form (e.g., 15 for MX RRs)	2
CLASS	Class code	2
TTL	Count of seconds that the RR stays valid (The maximum is $2^{31}-1$, which is about 68 years)	4
RDLLENGTH	Length of RDATA field (specified in octets)	2
RDATA	Additional RR-specific data	Variable, as per RDLLENGTH

Figure 28: DNS Response Resource Record. Source: wikipedia.org

Moreover, a DNS response includes two other sections: the authority section and the additional section. The **authority section** contains resource records pointing to authoritative name servers that are authoritative for the queried domain. By utilizing this section, the resolver can be properly directed if the complete answer was not given initially. While, the **additional section** may contain additional helpful information, such as IP addresses of the authoritative name servers listed in the authority section.

4.7 Proposed DNS-Based Detection Approach

Considering enterprise network security, an imperative question comes up: **why is it so important to introduce a DNS-based detection mechanism within our internal infrastructure?** This is a significant aspect of our approach to handling internal threats and providing high-end network security. Hence, interpreting the pertinence of such measures is central to dealing with the challenges of internal threats and ensuring robust network security. A **DNS-based detection algorithm** is crucial for securing internal networks by identifying and neutralizing potential breaches. It provides an essential layer of defense against both internal and external threats, significantly enhancing the organization's security posture. This approach has become a key advancement in detecting and preventing network scanning attempts and attacks.

As previously discussed in Chapter 3, traditional methods like signature-based and behavior-based detection have been widely used to identify malicious activities. In spite of that, these methods are not without downsides, especially in situations where the attackers use new techniques, obfuscation, or stealth attacks that do not fit into the established patterns.

Tackling the problem of network scanner detection is quite a challenge due to the attackers using advanced evasion techniques for getting through the detection systems. A frequent trick that attackers adopt is to cut down the scanning rate drastically, for instance, conducting a scan at a rate slower than one per minute. Such a low scanning frequency can be a good way to escape detection from many behavior-based algorithms that do not take into account such low-frequency activities as a scan, but rather consider them as normal background noise. These algorithms generally focus on traffic patterns and thresholds to recognize suspicious behavior, and slow scans can easily be mistaken for normal or benign traffic. Consequently, the detection systems may not be triggered to send alerts, or even if they do, the activity may be considered harmless, thus providing the attackers with a green light to obtain information about the network or systems without the risk of being detected.

On the other hand, DNS-based detection brings in a more dynamic and proactive perspective by taking advantage of the fact that DNS traffic is ubiquitous and essential in most modern networks.

DNS detection techniques are based on the fact that scanning activities can be uniquely identified by correlating DNS queries with connection attempts. For example, Whyte et al. [29] uploaded their method to detect network worms by linking requests to preceding DNS queries. This approach treats a connection without prior DNS activity as anomalous, which is a very effective way of finding network worms that do not conduct DNS lookups before establishing connections.

In the same way, Ahmad et al.'s NEDAC (NEtwork and DAta link layer Countermeasure) approach [30] is to a certain extent sending such DNS activities and marking those new connections when there is no preceding DNS query. NEDAC employs a threshold-based tactic where the nonexistence of a DNS lookup before multiple connection attempts is considered suspicious, and it may be a sign of the propagation of worms.

However, both approaches do not take into consideration the diminishing of Time-To-Live (TTL) values in the DNS responses, which is one of the main components in minimizing the false negatives when detecting network scanners. Usually, TTL values of DNS responses are fixed for a long period of time, i.e. 24 hours, which gives a machine the opportunity to gather a big DNS cache of records that map many internal and external addresses. Such a large cache can be exploited by attackers: if they compromise a machine with an extensive DNS cache, they can conduct scans from this machine on the IP addresses stored in the cache without triggering detection systems that do not monitor DNS cache activity.

The proposed method uses one of the TCP/IP networks' main features: for a typical communication usually starting from a local network machine to a remote IP address, there must be a previous DNS query to get the IP address itself (think of resolving a domain name). Scanning is the term used to describe a connection that does not go through this process.

Scanners and network worms, in fact, are usually engaged in scanning IP blocks, a technique whereby they send a probe or exploit data packets to a range of randomly selected IP addresses within the targeted network space without prior DNS lookups. This method, called **random block scanning** [32], requires selecting IP addresses randomly within a particular block or range. Attackers exploit this to actively investigate a whole network segment to locate hosts, services, and possible vulnerabilities. Random block scanning is one of the most reliable methods for detecting unprotected or improperly configured devices. This process usually kicks off with the picking of a base IP address and a subnet mask, which together define the set of valid IP addresses within the targeted block. Consider an IPv4 that consists of a 32-bit space address: an attacker can find a /24 subnet, which allows them to scan up to 256 IP addresses. The scanner then creates a list of IP addresses within this range and transmits probe packets to the addresses, analyzing the responses to detect active hosts and services. This methodology can be applied with the help of automation tools like Nmap, which have the capabilities to execute detailed scans of large IP ranges.

Furthermore, according to Jafar Haadi Jafrian et al. in their research paper "Detecting Network Scanning Through Monitoring and Manipulation of DNS Traffic" [31], the relevant DNS record must still be valid (i.e., not expired) during the connection setup, as its TTL dictates so.

Therefore, reducing the TTL value is crucial because if a machine still holds an unexpired DNS record for an IP address, a scanner operating from that machine (compromised) could target the IP without being detected. This is because the scanner would be using the cached DNS record, as the compromised machine had performed a DNS lookup previously and stored the records, thus avoiding any new DNS queries that might be monitored. By shortening the duration for which DNS records are cached, it is possible to minimize the likelihood of a scanner utilizing these entries to perform scans on IP addresses stored in the cache.

Consequently, to enhance detection accuracy and prevent evasion, Jafar Haadi Jafrian et al. introduced a system in their research, called **SDS (scan-detection-system)**, that intercepts DNS responses, reduces the TTL values of these responses to shorter durations, such as 60 to 300 seconds, and mandates that all connection attempts (by observing, for example, a TCP SYN segment, the first UDP datagram, the first ICMP packet) start with a valid DNS query. Any connections with no or expired preceding DNS lookup are treated as possible scans, which can be either recorded for further investigation or denied, depending on the network's security policies.

Moreover, to distinguish legitimate and malicious network traffic, it is possible to keep a list, configured during the setup phase, of allowed IP addresses that are excluded from the standard security policy to make sure that the essential services and internal communication (like mail servers) are not wrongly flagged. A connection attempt from a machine to a non-allowlisted IP without a prior DNS query is marked as a potential scan.

The SDS are strategically positioned at network boundaries, such as the entry points of LANs or VLANs (see Figure 29).

The decentralized architecture of this system, with each subnet having its own SDS units that function independently, makes it more scalable and fault-tolerant. Thus, it can be deployed in simple or complex networks of different sizes.

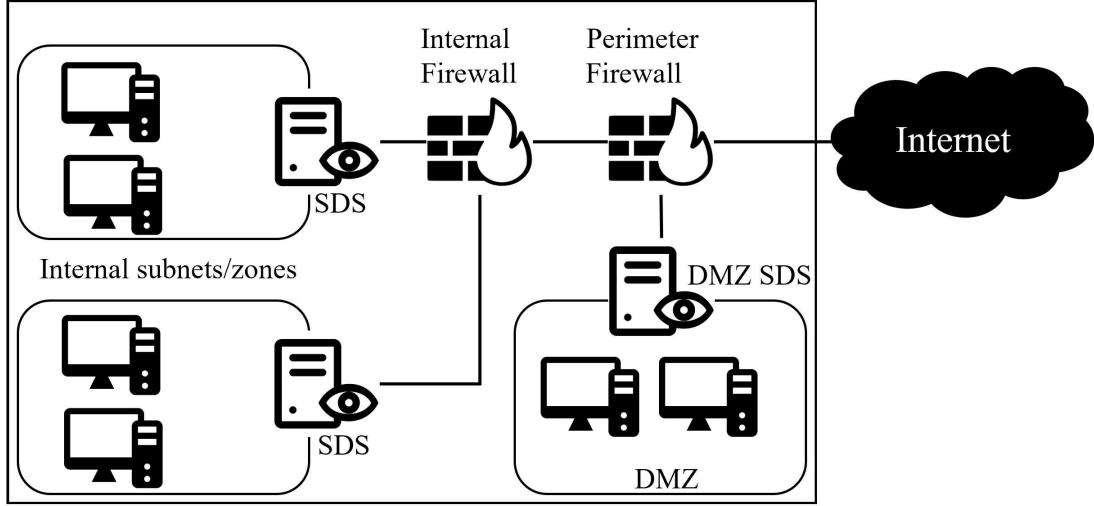


Figure 29: Position of SDS in an Enterprise Network Example. [31]

Below is the SDS algorithm presented in pseudocode (see Figure 30).

Algorithm 1 SDS Algorithm for Detecting Internally-Initiated Scans

```

%marking origins of DNS queries
for every ingress DNS response destined to a machine
with address  $IP_S$  do
    for every type-A RR  $\in$  DNS response providing
address  $IP_D$  do
         $ht[IP_S||IP_D] \leftarrow (clock + \tau)$ 
        update TTL of DNS RR to  $\tau$ 
        update UDP checksum of DNS response
        forward response
%validating connections
for every new egress connection from  $IP_S$  to  $IP_D$  at time
 $t$  do
    if exists( $ht[IP_S||IP_D]$ ) and  $ht[IP_S||IP_D] \geq t$  then
        allow connection
    else
        drop/log connection
        block  $IP_S$  (if policy allows)

```

Figure 30: SDS Algorithm for Detecting Internally-Initiated Scans. [31]

These are the main features of the SDS algorithm:

- **DNS response manipulation:** SDS intercepts DNS responses destined for internal machines, and it modifies the TTL values (denoted by τ) of all type-A DNS Resource Records (RRs) for IPv4 to short durations (e.g., 60-300 seconds). Then, the SDS recalculates the

UDP header checksum and forwards the DNS response. In this way, the SDS ensures that machines cache only the most recent DNS records, thus preventing the use of outdated cached IP addresses for scanning.

- **Detection of unauthorized connections:** The SDS system detects outbound internal scans targeting external IP addresses (local-to-remote scans). It maintains a hash table of DNS responses, where the key is a combination of the source and destination IPs ($IPs || IPd$) and the value includes their expiration times plus the clock timestamp. The system then verifies if the connection attempt was preceded by a valid DNS lookup within the adjusted period. Connections that lack a corresponding recent DNS query are flagged as potential scans.
- **Local processing and scalability:** SDS operates within the local subnet (LAN/VLAN) without requiring global clock synchronization. Each SDS device is responsible for its own subnet, enhancing scalability and simplifying deployment.
- **Flexible response actions:** Upon detecting a scan, the SDS can either drop the connection, log it for further inspection, or both, based on predefined security policies. It also allows for additional actions such as denylisting or quarantining the scanning machine if required.

Introducing an SDS device that implements the aforementioned algorithm into an enterprise network infrastructure is crucial for several key reasons:

- Detection of malicious internal activity: internal users, whether employees or possibly infected machines, could represent a massive security threat. These insiders could conduct a network scan intentionally for vulnerabilities or sensitive information. An internally focused DNS-based detection algorithm can identify abnormal behaviors indicative of scanning activities, such as a connection attempt without a clear purpose, by monitoring DNS queries and correlating them with connection attempts. This is crucial for blocking unauthorized access and recognizing possible internal dangers in time before they grow into more serious security incidents.
- Identifying compromised devices: devices that are part of the network can be broken into and used as the initial place for the next attack. A machine that has been hacked can be set to scan other internal systems or even external networks. DNS-based detection can help identify such activities by noting connections that occur without a prior DNS query, which is not the case with normal traffic. Through this, security teams are able to quickly cut off and fix the compromised devices, thereby limiting the chance of data breaches and other attacks.

- Preventing external threats and reducing attack surface: scans that are initiated internally can often very well be a sign of the possibility of attacks on external targets. By detection and mitigation of such activities, organizations can prevent their networks from being used as a launch pad for attacks on other entities, thus reducing their own liability and preserving their reputation. This is particularly vital in environments where compliance with regulatory requirements is strict.
- Complementing other security measures: DNS-based detection algorithms do work well with other security measures like firewalls, several detection algorithms implemented on intrusion detection and prevention systems, and endpoint protection. While the other tools are critical, they may not always be able to identify threats that come from inside the network. A DNS-based method takes a different angle by looking at the first steps of communication, which are DNS lookups, and thus provides another security layer.

Chapter 5

Case study

In this chapter, we delve into the practical application of the proposed DNS-based detection algorithm within a controlled laboratory environment. The setup involves the configuration of three virtual machines to simulate an enterprise network, detailing the software specifications, the network architecture, and the specific Python scripts that implement the detection algorithm outlined in the previous chapter, as well as a script for automated browsing. This structured approach ensures a comprehensive understanding of the algorithm's effectiveness in real-world scenarios.

5.1 Laboratory Setup

To demonstrate the practicality of the proposed DNS-based detection method, we constructed a test environment featuring a small network setup with network address 192.168.100.0/24. We utilized VirtualBox 7.0.20 to create three virtual machines (VMs), each serving a specific role, all running Ubuntu LTS 6.8.0-31-generic x86_64 GNU/Linux.

The virtual machines are configured as follows:

- **VM1 - Benign Client:** this machine simulates regular user activity by using a Python script with Chrome Selenium to automatically browse the top 1,000,000 websites sourced from a curated list. Specifically, these domains are derived from the "top-1000000-domains" list, available on GitHub at [zer0h/top-1000000-domains](https://github.com/zer0h/top-1000000-domains) [33].

This list includes a variety of popular websites across different categories, ensuring a broad and realistic simulation of typical user browsing behavior.

The network settings for this VM are set to "Connected to: Internal Network. Name: intnet".

The network interface is configured as follows (see Figure 31):

```

enp0s8: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
inet 192.168.100.3 netmask 255.255.255.0 broadcast 192.168.100.255
inet6 ::::: prefixlen 64 scopeid 0x20<link>
ether ::::: txqueuelen 1000 (Ethernet)
RX packets 0 bytes 0 (0.0 B)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 114 bytes 10954 (10.9 KB)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

```

Figure 31: VM1 Network Interface Configuration.

The network interface enp0s8 is configured with the IP address 192.168.100.3 and has a subnet mask of 255.255.255.0. This setup ensures that VM1 operates within the subnet 192.168.100.0/24, which allows for communication with other devices (VM2 and VM3) on the same local network.

To ensure that all traffic flows through VM3, which acts as the default gateway, we configured the kernel IP routing table for VM1 as follows (see Figure 32):

Kernel IP routing table							
Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
0.0.0.0	192.168.100.10	0.0.0.0	UG	20100	0	0	enp0s8
192.168.100.0	0.0.0.0	255.255.255.0	U	100	0	0	enp0s8

Figure 32: VM1 Kernel IP Routing Table Configuration.

The default gateway is set to 192.168.100.10, which is the IP address of VM3 on the internal network (the flag UG, on the first line, indicates that the route is up and used as gateway).

By modifying the VM1's DNS settings, we can set the nameservers to be 8.8.8.8 and 8.8.4.4 (Google public DNS servers).

Since VM3 is set to be the default gateway, this configuration ensures that all outbound traffic from VM1 is routed through VM3 before reaching external networks, and it allows that all the DNS queries made by VM1 will also be directed to VM3.

Moreover, this setup allows VM3 to monitor and intercept DNS responses, providing the necessary data for the scan detection system.

Here is the Python script for VM1, which is responsible for automated browsing activity:

```

from selenium import webdriver
from selenium.webdriver.chrome.options import Options
from selenium.webdriver.chrome.service import Service
from webdriver_manager.chrome import ChromeDriverManager

```

```

import time

options = Options()
options.add_argument('--headless')
options.add_argument('--no-sandbox')
options.add_argument('--disable-dev-shm-usage')
driver = webdriver.Chrome(service=Service(ChromeDriverManager().install
                                ()), options=options)

# Define websites list
website_list = [
    "https://www.google.com",
    "https://www.wikipedia.org",
    "https://www.youtube.com",
    "https://facebook.com",
    "https://instagram.com",
    # Add more websites to the list here (cut short to avoid exceeding
    # the length)
]

try:
    for website in website_list:
        driver.get(website)
        print(f"Browsing: {website}")
        time.sleep(2) # wait time between browsing websites
except Exception as e:
    print(f"Error browsing {website}: {e}")
finally:
    driver.quit()

```

The script utilizes the Selenium library, a powerful tool for automating web browsers. Specifically, it employs the Chrome WebDriver, which is managed using the `webdriver_manager` package to ensure the correct version of the driver is used.

Therefore, the script runs Chrome in headless mode (*-headless*), meaning the browser operates without a graphical user interface. This is efficient and suitable for automated tasks as it reduces resource consumption. Additional options like *-no-sandbox* and *-disable-dev-shm-usage* are included to ensure smooth execution in various environments, including those with limited resources.

The script includes a predefined list of websites, representing common and popular destinations on the internet. These URLs are sourced from well-known sites like Google,

Wikipedia, and YouTube, among others. This selection is intended to mimic a realistic browsing session of an average user. The script iterates over the list of websites, navigating to each one sequentially. It utilizes Selenium’s *get* method to load each website.

A *time.sleep(2)* command is employed to introduce a brief delay between each browsing action. This delay simulates a more natural browsing pace, avoiding rapid requests that might otherwise be flagged as abnormal or suspicious by the websites being visited or by some network monitoring tool.

The script also includes a *try-except* block to handle potential exceptions that might occur during the browsing process, such as network issues or page load errors. Errors are caught and printed, ensuring that the script can continue to the next website in the list without interruption.

In the *finally* block, the script ensures that the ChromeDriver instance is properly closed with *driver.quit()*. This step is crucial to free up system resources and avoid leaving hanging processes.

The automated browsing script serves multiple critical roles in the experiment; for example, it generates legitimate DNS queries and responses that can be monitored and analyzed by the SDS (Scan Detection System) implemented on VM3.

This helps in distinguishing between legitimate and potentially harmful DNS activities.

- **VM2 - Scanner:** this VM simulates a malicious entity by performing scanning activities on specific external IP addresses using the Nmap tool (version 7.94). It shares the same network settings as VM1 to ensure it is part of the same internal network.

The network interface is configured as follows (see Figure 33):

```
enp0s8: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.100.4 netmask 255.255.255.0 broadcast 192.168.100.255
    inet6 ::::: prefixlen 64 scopeid 0x20<link>
    ether ::::: txqueuelen 1000 (Ethernet)
    RX packets 342 bytes 323385 (323.3 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 286 bytes 26144 (26.1 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

Figure 33: VM2 Network Interface Configuration.

The network interface enp0s8 is configured with the IP address 192.168.100.4 and has a subnet mask of 255.255.255.0.

As for VM1, to ensure that all traffic flows through VM3, we configured the kernel IP routing table for VM2 as follows (see Figure 34):

Kernel IP routing table							
Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
0.0.0.0	192.168.100.10	0.0.0.0	UG	100	0	0	enp0s8
192.168.100.0	0.0.0.0	255.255.255.0	U	100	0	0	enp0s8

Figure 34: VM2 Kernel IP Routing Table Configuration.

The default gateway is set to 192.168.100.10 (VM3's IP address).

Moreover, as we did for VM1, we modified the VM2's DNS settings and used Google public DNS servers as nameservers (8.8.8.8 and 8.8.4.4). In this way, as we previously mentioned for VM1, all the DNS queries and responses made by the VM2 will be directed to VM3.

To implement the scanning activities on VM2, we have configured it to run Nmap against a predefined list of IP addresses. These IP addresses were resolved from domain names of vulnerable web applications listed on this GitHub repository [34]. By using IPs associated with these domains, we can ethically perform our scanning without violating any legal or ethical boundaries. This setup simulates the behavior of a scanner targeting external IPs (for example, by using the random block scanning technique), which aligns with our experimental focus.

We chose to scan these specific external IPs instead of internal ones because, in an internal network, Nmap would typically use the ARP protocol to resolve IP addresses of machines within the same subnet, bypassing DNS queries. Our goal is to demonstrate the detection of scanning activities targeting external IPs, leveraging DNS-based detection mechanisms. Below is an example of an Nmap command executed on VM2:

```
nmap -p 1-1024 -T4 -A -v -iL /home/scannervm2/Desktop/listipscan.txt
```

This command uses Nmap to scan ports 1 to 1024 on the target hosts, with a timing and speed of the scan to level 4 (aggressive), including OS detection, version detection, script scanning, and traceroute, against a pre-defined list of IPs.

The "listipscan.txt" contains these IP addresses:

```
44.228.249.3
44.238.29.244
52.16.210.4
216.58.209.52
34.249.203.140
216.58.204.148
188.114.97.7
54.82.22.214
```

```
65.61.137.117
54.73.53.134
69.164.223.171
69.164.223.208
54.204.37.212
107.20.213.223
52.1.112.43
192.124.249.5
178.79.134.182
```

The scanning activities will help us monitor how VM3 (the SDS machine) detects these scans based on the absence of preceding DNS queries.

- **VM3 - SDS (Scan Detection System):** this VM acts as the detection system in our setup, monitoring and analyzing DNS traffic to identify potential scans. It is equipped with two network interfaces: the first, like VM1 and VM2, is set to "Internal Network" (intnet), connecting it to the virtual internal network, while the second interface is set to "Bridged Adapter", allowing it to connect to the physical network and the Internet. VM3 also functions as the default gateway for VM1 and VM2, routing all network traffic through it.

Like for VM1 and VM2, VM3 uses Google public DNS servers (8.8.8.8 and 8.8.4.4) as nameservers in its DNS configuration file.

A Python script continuously runs on this VM to monitor, intercept, modify, and forward network traffic as needed.

5.2 Detection Algorithm Implementation

In this section, we present the Python script executed on VM3 that serves as a practical implementation of the proposed algorithm. The script is designed to actively monitor network traffic, intercept and inspect packets, and modify them when necessary before forwarding them to their intended destination. By implementing a series of checks and filters, it ensures that only legitimate and secure communications are allowed, thereby enhancing the overall security of the network.

The Python script implementing the Scan Detection System (SDS) can be found in Appendix A. This script tracks DNS responses, alters their Time-to-Live (TTL) values, and logs incoming connections, identifying those without a preceding and valid DNS lookup as scanning attempts, by leveraging the netfilterqueue and scapy libraries. For this experiment, the TTL for DNS responses was set to 300 seconds (5 minutes), ensuring that DNS cache entries are refreshed

frequently to mitigate potential evasion tactics by scanners.

The overarching goal is to enhance network security by controlling DNS traffic, tracking suspicious activities, and managing IP addresses through dynamic blocking and expiration mechanisms.

In the following sections, we will discuss the libraries and modules, data structures, the global variables, and core functions used in the script, providing detailed descriptions of their roles and how they contribute to the overall implementation.

Libraries and Modules

- **netfilterqueue**: this module serves as a bridge between the Linux Netfilter framework and user-space programs, allowing packets to be intercepted, inspected, modified, and reinjected back into the kernel. In the script, `netfilterqueue` binds the packet processing function (`process_packet`) to the network traffic, enabling real-time packet filtering and manipulation. Moreover, `iptables`, a command-line utility used in Linux to configure the Netfilter framework, is employed to set up rules that redirect traffic to the `netfilter` queue, allowing the Python script to process packets. In fact, the `iptables` command used configures the firewall to send specific network traffic to the user-space queue, where it can be inspected and modified as needed.
- **scapy**: it is a powerful Python library used for packet manipulation, enabling the creation, modification, and analysis of network packets at various layers. It supports numerous protocols and allows users to craft and manipulate packets with great flexibility. In this script, `scapy` is utilized to parse and inspect packets, extract and modify DNS records, and manage other network-related tasks such as identifying suspicious connections and handling DNS queries and responses.

Moreover, the `scapy_packet = scapy.IP(packet.get_payload())` line in the script extracts the intercepted packet's payload and constructs it into a Scapy IPv4 or IPv6 packet object (depending on the version of the IP protocol).

After extracting the payload using `packet.get_payload()`, Scapy allows for easy parsing of various network layers like IP, TCP, DNS, and more. For example, `scapy_packet[scapy.DNS]` is used in order to extract DNS layer information and access DNS fields and records.

- **dns and IPv6** : these submodules from `scapy.layers` are specifically imported to work with DNS and IPv6 traffic.
- **time, pickle, os, psutil, socket, subprocess, threading, queue as queue_module**: these are standard Python libraries used for various system operations, including time

tracking, serialization (saving and loading objects), system interactions, threading, and inter-thread communication.

Data Structures

- **dns_expiration_table**: this is a hash table (dictionary) that stores the source IP, the DNS response-provided IP, and an associated Time-To-Live (TTL) value. It is used to track the validity of DNS records and ensure that stale or unauthorized connections are blocked or dropped. This table plays a crucial role in maintaining the integrity and security of DNS communications by allowing only timely and legitimate connections.
- **allowlisted_users**: this list contains IP addresses of users that are explicitly trusted within the network. Traffic originating from or directed to these IP addresses is automatically accepted without being subject to the same scrutiny as other traffic. By maintaining this allowlist, the script ensures that known and trusted entities are not incorrectly flagged as suspicious, improving both security and performance.
- **allowlisted_domains**: Similar to allowlisted users, this list contains domain names that are considered trusted. DNS queries and responses (also reverse-DNS lookups) involving these domains are allowed through without extensive filtering. In order to prevent false positives in the detection system, we added certain IP addresses to this data structure. This was necessary because Ubuntu performs reverse DNS queries as part of its internet connectivity checking process. By excluding these IPs from detection, we ensure that these legitimate system operations do not trigger unnecessary alerts or interfere with the scanning detection system's functionality.
- **allowlisted_protocols**: this list specifies protocols that are allowed to communicate freely across the network, such as ICMP (for ping) and SSH (for secure shell connections). Traffic using these protocols is exempt from some of the more stringent checks applied to other types of traffic, allowing essential network operations to proceed unhindered.
- **blocked_ips**: it is a dictionary that keeps track of IP addresses that have been flagged as malicious and are therefore blocked from further communication. Each entry is associated with an expiration time, after which the block will be lifted unless the IP is found to be suspicious again. This structure helps in dynamically managing the network's security by temporarily isolating potential threats.
- **suspicious_count**: it is a dictionary that records the number of suspicious activities associated with each source IP address. When a certain threshold is reached, the associated IP is blocked (it is added to the blocked_ips dictionary). This data structure enables the

script to implement a form of behavioral analysis, where repeated suspicious actions trigger stricter security measures, such as blocking the offending IP.

Global Variables

- **Performance Metrics:** variables like `latencies`, `cpu_usages`, `memory_usages` and `packet_count` are used to track the performance of the script during its execution (they are printed on screen every 10000 processed packets).
- **DNS Expiration File:** this file is used to persistently store the DNS expiration table's contents. The DNS expiration table, which tracks the validity of DNS records by storing IP mappings and their expiration times, is periodically saved to this file. This ensures that the state of the DNS expiration table is preserved across script restarts or system reboots. When the script is initialized, the file is loaded to restore the DNS expiration table, allowing the script to continue monitoring and filtering DNS traffic based on the previously recorded data. This mechanism is critical for maintaining consistent network security, avoiding false positives, and minimizing downtime caused by potential attacks or disruptions.

Core Functions

- **`set_iptables_rule()`:** it establishes an iptables rule to direct all forwarded packets to a netfilter queue for processing.
- **`empty_dns_expiration_table()`:** this function ensures the DNS expiration table is cleared, effectively removing all stored DNS records from the system. By calling this function, all cached DNS entries are deleted, forcing the system to refresh DNS information from scratch. This can be useful during the initialization of the script or when a reset of DNS data is necessary for security or troubleshooting purposes. Ensuring that outdated or incorrect DNS records are purged helps maintain the accuracy and security of DNS resolution.
- **`save_dns_expiration_table()`:** this function is responsible for saving the current state of the DNS expiration table to the DNS Expiration File. The function ensures that all active DNS records, along with their respective expiration times, are serialized and written to a persistent storage file. This process is critical for maintaining continuity between script sessions as it allows the script to resume its operations without losing track of DNS records

that are still valid. By saving this information, the system can recover quickly after reboots or interruptions, maintaining the integrity of the DNS filtering process.

- **load_dns_expiration_table()**: it loads the DNS expiration table from the DNS Expiration File upon the script's initialization. By reading and deserializing the stored data, the function restores the DNS expiration table to its previous state, allowing the script to continue monitoring and filtering DNS traffic seamlessly.
- **get_source_ip(packet)** and **get_destination_ip(packet)**: these functions are fundamental for extracting the source and destination IP addresses from a network packet, considering both IPv4 and IPv6 formats. *get_source_ip(packet)* retrieves the IP address from which the packet originated, while *get_destination_ip(packet)* identifies the packet's intended recipient. These IP addresses are vital for the script's decision-making processes, such as determining whether a packet should be allowed or blocked based on its origin or destination. Accurate extraction of these IP addresses enables the script to apply its security policies effectively.
- **get_rdata_records(packet)**: Extracts the value of the field "rdata" of the DNS resource records from the DNS response packet, focusing on A (IPv4) and AAAA (IPv6) record types. "rdata" refers to the resource data that contains the actual information returned by the DNS query, such as the IP address associated with a domain name. This function checks for the presence of these records in the packet. If rdata records are found, the packet may need to be modified before it is forwarded to its destination. In the script's *process_packet(packet)* function, this check is essential because it determines whether the DNS response should be altered using the *modify_dns_response(scapy_packet)* function. Specifically, if rdata records are present, the response might need modification in order to change the TTL value. However, if no rdata records are found, there's no need for modification, and the packet can be accepted as is. This logic ensures that only relevant DNS responses are altered, optimizing the script's performance.
- **modify_dns_response(packet)**: it is tasked with altering the DNS response contained in the packet. After analyzing the rdata records or other parts of the DNS response, the script might decide to modify the response before it reaches the client. Hence, it is used to modify the DNS response's TTL field and updates the *dns_expiration_table* to track the TTL expiration time.
- **is_reverse_dns_query(packet)**: this function checks whether the given packet is a reverse DNS query (PTR record), which is a request to resolve an IP address back to its corresponding domain name. Identifying reverse DNS queries is important because these types of queries can sometimes be used for malicious purposes, such as reconnaissance

activities by an attacker. By detecting reverse DNS queries, the script can apply specific security measures, such as allowing or blocking the query based on whether it comes from a trusted source or not.

- **is_allowlisted_reverse_dns(packet)**: this function determines whether a reverse DNS query is from an allowlisted source. It checks the packet against a list of trusted IP addresses or domains that are permitted to perform reverse DNS lookups. If the query is allowlisted, it is considered safe and is allowed to proceed; otherwise, it may be blocked to prevent potential abuse.
- **check_reverse_dns_query(packet)**: this function validates if a reverse DNS query matches an existing record in the *dns_expiration_table*, determining its legitimacy based on previously established connections. Specifically, the *check_reverse_dns_query(packet)* function plays an essential role in the script's security mechanism because attackers may attempt to bypass the scan detection algorithm by exploiting reverse DNS queries. In fact, in the implemented algorithm, a connection is typically flagged as malicious if there is no preceding DNS lookup recorded in the DNS expiration table, indicating that the source IP is attempting to establish a connection without first resolving the destination IP's domain name (an action characteristic of scanning behavior). However, a legitimate connection is generally accompanied by a prior forward DNS lookup, and these DNS data are recorded in the script's DNS expiration table. An informed attacker who understands this detection logic might try to evade the system by first issuing reverse DNS queries to gather domain names associated with potential targets. After obtaining the domain name via the reverse DNS query, the attacker could then perform a forward DNS lookup for that domain. This action would create an entry in the DNS expiration table, falsely indicating that the connection is legitimate. The attacker could then proceed with scanning the IP addresses without triggering the detection system, as the required DNS lookup would appear to have previously been performed legitimately. Thus, *check_reverse_dns_query(packet)* function is important to identify and mitigate this type of evasion attempt, ensuring that the system remains robust against more sophisticated attackers who might try to exploit reverse DNS queries to bypass the detection algorithm.
- **calculate_metrics(start_time)**: it is responsible for monitoring and reporting the performance of the scan detection system. This function performs a series of tasks aimed at measuring key performance indicators (KPIs) such as latency, CPU usage, and memory usage.
- **process_commands()**: this function runs in a separate thread, providing a command-line interface for real-time interaction with the script. It continuously listens for user input and

executes various commands that allow the user to manage the network monitoring process. Commands include operations such as blocking or unblocking IP addresses, adding or removing allowlisted users or domains, printing current lists (e.g., blocked IPs, allowlisted users), and displaying the current DNS expiration table.

- **show_help()**: this function displays a list of available commands that can be used within the command-line interface. It provides descriptions of each command, helping the user understand how to interact with the script effectively. The *show_help()* function is particularly useful for new users or when a user needs a reminder of the available command options.
- **process_packet(packet)**: it is responsible for analyzing, modifying, and filtering network packets in real-time, based on various criteria such as DNS queries, DNS responses, IP blocking, and traffic handling. We will delve deeper into this function in the next paragraph, where we will provide a detailed explanation of its implementation choices and the rationale behind them.

5.3 Understanding the Core Logic: The process_packet Function

The **process_packet(packet)** function is the main component of the network scanning detection system implemented in the script. Its primary role is to analyze incoming network packets, make real-time decisions based on predefined security rules, and take appropriate actions such as modifying or blocking traffic.

The function begins by incrementing the global packet counter and by starting the timer that will be used for measuring the latency involved in processing each packet. This ensures that performance metrics are consistently recorded, allowing for subsequent analysis of CPU and memory usage. Throughout its execution, the function continuously updates performance metrics, printing summary statistics after processing every 10,000 packets.

Upon receiving a packet, the function first determines whether it is an IPv4 or IPv6 packet and proceeds to handle expired blocked IP entries by purging them from the `blocked_ips` dictionary. This step is crucial for maintaining an up-to-date list of blocked IPs and ensuring that stale entries do not inadvertently hinder legitimate traffic.

The function then inspects the packet to identify whether it contains a DNS response (DNSRR), a DNS query (DNSQR), or a reverse DNS query.

For **DNS responses**, it invokes the *get_rdata_records(packet)* function to check for any RDATA records, which are essential for determining if the DNS response should be modified. If RDATA records are present, the packet is modified through the *modify_dns_response(packet)* function,

and the verdict to accept or drop the packet is determined accordingly. This logic is critical, as it prevents unnecessary modifications to DNS responses when no RDATA records are found, optimizing performance and ensuring accuracy in DNS handling.

In the case of **reverse DNS queries**, the function employs *check_reverse_dns_query(packet)* to detect potential evasion techniques used by attackers. This step is particularly significant in the context of the script's overall security strategy. If an attacker attempts to bypass the scan detection algorithm by conducting reverse DNS lookups before performing scans, the script will identify such behavior and block the malicious activity. By allowing only legitimate reverse DNS queries through the *is_allowlisted_reverse_dns(packet)* function, the system mitigates the risk of false positives while maintaining robust protection against sophisticated scanning techniques.

For **standard DNS queries**, the function allows the packets to pass without further examination, given that such queries are generally benign and essential for normal network operations. Finally, the function examines **non-DNS traffic** by comparing the source and destination IPs against the *dns_expiration_table*. If a valid DNS association exists, the packet is accepted; otherwise it undergoes further checks to determine if it involves allowlisted protocols or applications (e.g., ICMP or SSH). Any connections judged suspicious are logged, and if a source IP exceeds a predefined threshold of suspicious activities (set to 100 in the experiment), it is blocked (for 1000 seconds as the *block_duration* variable in the script dictates so) to prevent further malicious attempts.

Chapter 6

Analysis and results

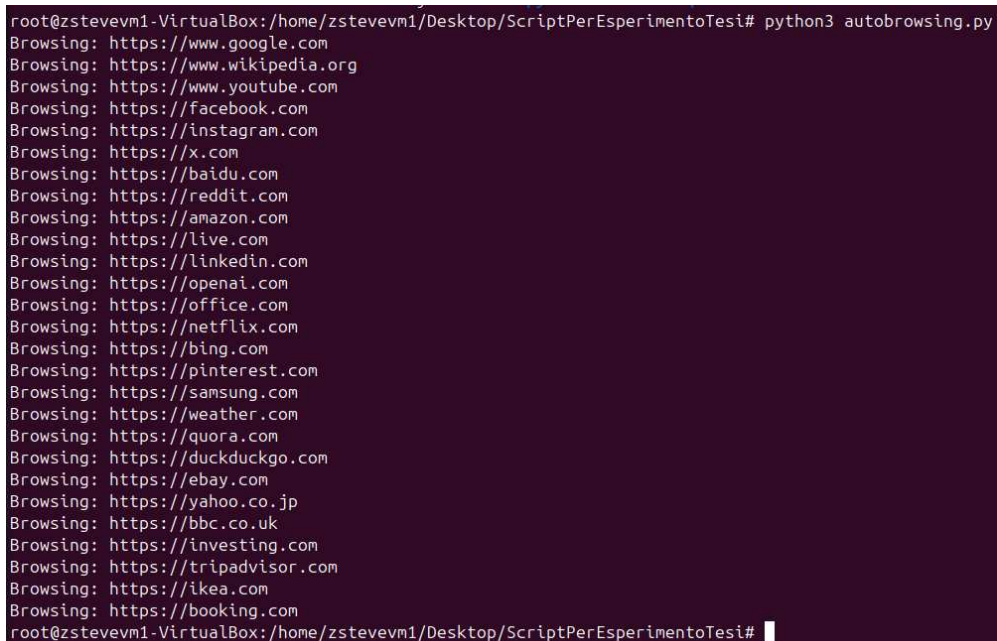
6.1 Experimental Execution and Results

This section details the design and implementation of the experiment aimed at evaluating the effectiveness of the scan detection system. The experiment involved three distinct virtual machines (VMs), each serving a unique role in the simulation, as described in the previous chapter. To begin the experiment, the scan detection system (SDS) script was executed on **VM3**, designed to monitor and block any suspicious network activity. The following command was used to initiate the script:

```
python3 sdsscript.py
```

In parallel, **VM1** was configured to generate legitimate network traffic, simulating typical user behavior through automated web browsing. This was achieved by running the autobrowsing script with the command:

```
python3 autobrowsing.py
```



```
root@zstevevm1-VirtualBox:/home/zstevevm1/Desktop/ScriptPerEsperimentoTesi# python3 autobrowsing.py
Browsing: https://www.google.com
Browsing: https://www.wikipedia.org
Browsing: https://www.youtube.com
Browsing: https://facebook.com
Browsing: https://instagram.com
Browsing: https://x.com
Browsing: https://baidu.com
Browsing: https://reddit.com
Browsing: https://amazon.com
Browsing: https://live.com
Browsing: https://linkedin.com
Browsing: https://openai.com
Browsing: https://office.com
Browsing: https://netflix.com
Browsing: https://bing.com
Browsing: https://pinterest.com
Browsing: https://samsung.com
Browsing: https://weather.com
Browsing: https://quora.com
Browsing: https://duckduckgo.com
Browsing: https://ebay.com
Browsing: https://yahoo.co.jp
Browsing: https://bbc.co.uk
Browsing: https://investing.com
Browsing: https://tripadvisor.com
Browsing: https://ikea.com
Browsing: https://booking.com
root@zstevevm1-VirtualBox:/home/zstevevm1/Desktop/ScriptPerEsperimentoTesi#
```

Figure 35: Autobrowsing script executed on VM1.

During its execution, the autobrowsing script on VM1 successfully navigated through the list of websites specified in its configuration, visiting each site sequentially and contributing to the background network traffic, as evidenced by the preceding screenshot (see Figure 35). Simultaneously, the scan detection system script running on VM3 was designed to operate silently in the background, avoiding printing real-time output to the terminal. However, it actively logged and processed incoming data. Specifically, it began modifying the TTL values to 300 seconds for all DNS Resource Records of type A and/or AAAA as part of its defensive measures against potential scans. Below is an example of a DNS response (Scapy packet) modified by the VM3's script, showcasing the adjusted TTL values:

```
###[ IP ]###
version    = 4
ihl        = 5
tos        = 0x0
len        = None
id         = 10273
flags      = DF
frag       = 0
ttl        = 122
proto      = udp
chksum     = None
src        = 8.8.8.8
dst        = 192.168.100.3
\options   \
###[ UDP ]###
sport      = domain
dport      = 43013
len        = None
chksum     = None
###[ DNS ]###
id         = 48008
qr         = 1
opcode     = QUERY
aa         = 0
tc         = 0
rd         = 1
ra         = 1
z          = 0
ad         = 0
cd         = 0
```

```

rcode      = ok
qdcount    = 1
ancount    = 9
nscount    = 0
arcount    = 0
\qd        \
|###[ DNS Question Record ]###
|  qname    = 'www.youtube.com.'
|  qtype    = A
|  qclass   = IN
\an        \
|###[ DNS Resource Record ]###
|  rrname   = 'www.youtube.com.'
|  type     = CNAME
|  rclass   = IN
|  ttl      = 171
|  rdlen    = None
|  rdata    = 'youtube-ui.l.google.com.'
|###[ DNS Resource Record ]###
|  rrname   = 'youtube-ui.l.google.com.'
|  type     = A
|  rclass   = IN
|  ttl      = 300
|  rdlen    = 4
|  rdata    = 216.58.204.142
|###[ DNS Resource Record ]###
|  rrname   = 'youtube-ui.l.google.com.'
|  type     = A
|  rclass   = IN
|  ttl      = 300
|  rdlen    = 4
|  rdata    = 142.251.209.14
|###[ DNS Resource Record ]###
|  rrname   = 'youtube-ui.l.google.com.'
|  type     = A
|  rclass   = IN
|  ttl      = 300
|  rdlen    = 4
|  rdata    = 216.58.209.46
|###[ DNS Resource Record ]###
|  rrname   = 'youtube-ui.l.google.com.'
|  type     = A

```

```

|   rclass      = IN
|   ttl         = 300
|   rdlen       = 4
|   rdata       = 142.251.209.46
|###[ DNS Resource Record ]###
|   rrname      = 'youtube-ui.l.google.com.'
|   type        = A
|   rclass      = IN
|   ttl         = 300
|   rdlen       = 4
|   rdata       = 216.58.205.46
|###[ DNS Resource Record ]###
|   rrname      = 'youtube-ui.l.google.com.'
|   type        = A
|   rclass      = IN
|   ttl         = 300
|   rdlen       = 4
|   rdata       = 142.250.180.174
|###[ DNS Resource Record ]###
|   rrname      = 'youtube-ui.l.google.com.'
|   type        = A
|   rclass      = IN
|   ttl         = 300
|   rdlen       = 4
|   rdata       = 216.58.204.238
|###[ DNS Resource Record ]###
|   rrname      = 'youtube-ui.l.google.com.'
|   type        = A
|   rclass      = IN
|   ttl         = 300
|   rdlen       = 4
|   rdata       = 142.250.180.142
ns      = None
ar      = None

```

Additionally, during its execution, the scan detection system script allows for user interaction through predefined commands entered into the terminal. For instance, while VM1 is engaged in browsing, the command *print_hash_table* can be used on VM3's script to display the current state of the DNS expiration table (see Figure 36).

```

Enter a command: print_hash_table

DNS Expiration Table:
192.168.100.3|2620:2d:4002:1::197: 2024-08-15 23:18:45
192.168.100.3|2620:2d:4000:1::23: 2024-08-15 23:18:45
192.168.100.3|2620:2d:4002:1::198: 2024-08-15 23:18:45
192.168.100.3|2001:67c:1562::23: 2024-08-15 23:18:45
192.168.100.3|2620:2d:4000:1::96: 2024-08-15 23:18:45
192.168.100.3|2620:2d:4000:1::2a: 2024-08-15 23:18:45
192.168.100.3|2620:2d:4000:1::22: 2024-08-15 23:18:45
192.168.100.3|2001:67c:1562::24: 2024-08-15 23:18:45
192.168.100.3|2620:2d:4000:1::98: 2024-08-15 23:18:45
192.168.100.3|2620:2d:4000:1::2b: 2024-08-15 23:18:45
192.168.100.3|2620:2d:4002:1::196: 2024-08-15 23:18:45
192.168.100.3|2620:2d:4000:1::97: 2024-08-15 23:18:45
192.168.100.3|185.125.190.49: 2024-08-15 23:18:45
192.168.100.3|91.189.91.48: 2024-08-15 23:18:45
192.168.100.3|185.125.190.96: 2024-08-15 23:18:45
192.168.100.3|185.125.190.97: 2024-08-15 23:18:45
192.168.100.3|185.125.190.17: 2024-08-15 23:18:45
192.168.100.3|185.125.190.18: 2024-08-15 23:18:45
192.168.100.3|91.189.91.96: 2024-08-15 23:18:45
192.168.100.3|91.189.91.98: 2024-08-15 23:18:45
192.168.100.3|91.189.91.97: 2024-08-15 23:18:45
192.168.100.3|185.125.190.48: 2024-08-15 23:18:45
192.168.100.3|91.189.91.49: 2024-08-15 23:18:45
192.168.100.3|185.125.190.98: 2024-08-15 23:18:45
192.168.100.4|91.189.91.97: 2024-08-15 23:17:50
192.168.100.4|91.189.91.48: 2024-08-15 23:17:50
192.168.100.4|185.125.190.97: 2024-08-15 23:17:50
192.168.100.4|91.189.91.98: 2024-08-15 23:17:50
192.168.100.4|185.125.190.17: 2024-08-15 23:17:50
192.168.100.4|185.125.190.48: 2024-08-15 23:17:50
192.168.100.4|185.125.190.18: 2024-08-15 23:17:50
192.168.100.4|185.125.190.49: 2024-08-15 23:17:50
192.168.100.4|91.189.91.49: 2024-08-15 23:17:50
192.168.100.4|91.189.91.96: 2024-08-15 23:17:50
192.168.100.4|185.125.190.98: 2024-08-15 23:17:50

```

Figure 36: *print_hash_table* command executed on VM3's script.

As observed, the DNS expiration table becomes populated as browsing activity increases, demonstrating the script's ongoing adjustments in response to network behavior.

In this experiment, **VM2**, according to its settings, was tasked with executing a network scan to simulate malicious scanning activities. The Nmap tool was utilized for this purpose, with the following command executed to scan a list of IP addresses across a range of ports:

```
nmap -p 1-1024 -T4 -A -v -iL /home/scannervm2/Desktop/listipscan.txt
```

Once the Nmap command was executed on VM2, the scan results clearly illustrated the impact of the scan detection system running on VM3. The scan, which under normal circumstances would complete much faster, encountered significant delays and complications due to the active interference from the detection system.

The Appendix B contains the full output from the Nmap scan performed by VM2 during the experiment, highlighting the extended scan times, increasing latencies, and the initial discovery of an open port before the detection system fully intervened.

The scan initially proceeded at normal speed, but as the detection system began dropping probes, the latency for each scan increased dramatically.

As the Nmap scan began on VM2, the scan detection system script running on VM3 was not only actively monitoring and filtering the incoming probes but also took additional measures to enhance network security. In fact, one of the key defensive mechanisms of this script is its ability to block reverse DNS queries that have not been preceded by a legitimate forward DNS query. Therefore, upon detecting these reverse DNS queries, the VM3 script intervened and blocked them, preventing Nmap from obtaining any meaningful reverse DNS information, and it invoked the function `check_reverse_dns_query(packet)`, which performs a series of checks. The script first determines if the DNS query is a reverse lookup by examining whether the query name ends with ".in-addr.arpa.", which is the domain used for reverse DNS lookups. It then extracts the actual IP address from the reverse query and checks if there was a legitimate forward DNS query corresponding to that IP address in the DNS expiration table. If a valid DNS lookup for the IP address is found in the table and the lookup is still within its TTL expiration period, the reverse DNS query is allowed. However, if no valid forward lookup exists, or if the TTL has expired, the reverse DNS query is blocked, preventing Nmap from discovering hostnames associated with its target IP addresses.

The behavior of the SDS script when interacting with VM1, which simulated regular user activity through the autobrowsing script, was different. Unlike VM2, VM1 performed legitimate forward DNS queries as it browsed the web. These DNS queries were captured and stored in the DNS expiration table by the SDS script. If VM1 were to initiate a reverse DNS query, the SDS script would follow the same process, checking the DNS expiration table for a corresponding valid forward lookup. If a valid forward DNS lookup had been made for the IP address, and the TTL had not expired, the reverse DNS query from VM1 would be accepted. This is because VM1's browsing activity generated legitimate DNS lookups, marking the resulting reverse queries as

safe. Below is a screenshot illustrating the blocked reverse DNS queries during the scan (see Figure 37).

```
=====
Reverse-DNS query from 192.168.100.4 detected. Blocking this dns reverse query.
=====
=====
Reverse-DNS query from 192.168.100.4 detected. Blocking this dns reverse query.
=====
=====
Reverse-DNS query from 192.168.100.4 detected. Blocking this dns reverse query.
=====
=====
Reverse-DNS query from 192.168.100.4 detected. Blocking this dns reverse query.
=====
=====
Reverse-DNS query from 192.168.100.4 detected. Blocking this dns reverse query.
=====
```

Figure 37: VM2's reverse-DNS queries blocked by VM3's script.

This screenshot clearly demonstrates the effectiveness of the scan detection system in identifying and blocking potentially harmful actions, further complicating the scanning efforts of VM2.

Furthermore, the Nmap output indicated that the SYN Stealth Scan, which typically completes quickly, took an unusually long time. Initial timing estimates predicted a completion time of about 16-17 minutes, but as the scan progressed, these estimates extended to several hours. Eventually, the scan took over 9300 seconds (approximately 2 hours and 35 minutes) to complete the scanning activity against 8 hosts only, with increasing delays in probe sending ("Increasing send delay" as displayed in the Nmap output) once VM3 detected and began filtering the traffic. This delay was not uniform across all IP addresses. For instance, the scan against IP 44.228.249.3 completed in 5347.22 seconds, while other IPs took even longer, such as 54.82.22.214, which took 8314.27 seconds. This indicates that the detection system was highly effective in slowing down

and eventually obstructing the scanning process.

```
*****
Suspicious connection from 192.168.100.4 to 216.58.209.52 detected. Count: 97
*****

*****
Suspicious connection from 192.168.100.4 to 34.249.203.140 detected. Count: 98
*****

*****
Suspicious connection from 192.168.100.4 to 216.58.204.148 detected. Count: 99
*****

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
Blocking IP 192.168.100.4 after 100 suspicious connections.

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

*****
Connection attempt from blocked IP 192.168.100.4 to 52.16.210.4 blocked.
*****
```

Figure 38: Threshold of 100 suspicious connections reached on VM3's script.

This initial discovery highlights a key aspect of the scan detection system: it does not block connections indiscriminately. The system allowed the scanner to send a limited number of probes, leading to the discovery of the open port. However, once the threshold was reached, VM3's detection system blocked VM2, preventing further meaningful reconnaissance (see Figure 39).

```
*****
Connection attempt from blocked IP 192.168.100.4 to 52.16.210.4 blocked.
*****

*****
Connection attempt from blocked IP 192.168.100.4 to 52.16.210.4 blocked.
*****

*****
Connection attempt from blocked IP 192.168.100.4 to 54.82.22.214 blocked.
*****

*****
Connection attempt from blocked IP 192.168.100.4 to 54.82.22.214 blocked.
*****

*****
Connection attempt from blocked IP 192.168.100.4 to 54.82.22.214 blocked.
*****
```

Figure 39: VM2's connections refused by VM3's script.

Overall, the entire scan took about 5 hours and 42 minutes (20,541.19 seconds), not only consuming an unexpectedly long amount of time due to effective obstruction by the detection system, but also providing no useful results: no open ports (apart from port 443 on host 54.82.22.214), operating system details, version detection, or other significant information were obtained.

6.2 Performance Analysis of the Scan Detection System

The scan detection system deployed on VM3, powered by an Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz, demonstrated highly efficient performance metrics during the experiment, even when faced with a substantial network load generated by over 1.3 million packets. The system maintained an impressive **average latency per packet** of just 0.001092 seconds, ensuring that the detection mechanisms operated swiftly without introducing significant delays in the network traffic. This low latency is crucial for maintaining the integrity of legitimate network communications while still effectively identifying and mitigating potential threats.

In terms of resource utilization, the VM3 was configured with 1 CPU operating with 1 core and 1 thread per core and 3.8 GiB of RAM. Despite this modest setup, the system maintained an **average CPU usage** of 8.72% and **average memory usage** of 33.68%. This indicates that the scan detection system is highly optimized, operating efficiently even under potentially heavy loads.

A notable aspect of the scan detection system's performance is the management of the DNS expiration table, which is a critical component of the system's defensive strategy. Despite handling a vast amount of DNS-related information, the expiration table's size remained remarkably compact. The **size of the *dns_expiration_table.pkl* file**, containing all the relevant DNS data, was just 32 KB. This small footprint underscores the effectiveness of the data structures and algorithms used in the script, allowing it to store and manage DNS records efficiently without consuming excessive disk space.

In summary, the scan detection system on VM3 demonstrated excellent performance across several key metrics, including latency, CPU and memory usage, and storage efficiency. The combination of low resource consumption and effective threat detection makes this system highly suitable for deployment in environments where maintaining network performance is as critical as ensuring security.

6.3 Limitations and Future Improvements

The scan detection system implemented on VM3 has really proven itself to be good at identifying and stopping the scanning activities, but there are still some drawbacks that should be taken into account for the next improvements.

The overload put on authoritative DNS servers is one significant limitation due to the lower TTL values for DNS Resource Records (RRs). Minimal TTL values lead to a rise in the number of DNS queries, which can be a burden for the authoritative DNS servers if they are not properly managed. This illustrates the importance of an **optimal TTL value** that gives the best detection but does not overload the server. The above parameter tuning is very important to avoid performance degradation of DNS infrastructure, especially in large networks.

Another challenge is the **identification of allowlisted addresses**, especially for machines and those IP addresses that come from other methods than DNS, such as static IP configurations or embedded IP addresses in HTTP payloads. This functionality is necessary because if the hosts are not identified properly, they may be blocked unnecessarily or recognized as false positives. In both cases, the normal network operations might be disrupted. For example, internal machines such as gateways, DNS servers, or devices like printers are essential to the network's functioning and must be allowlisted to avoid any service interruptions.

Knowing which hosts to allowlist is of course important for keeping the network working properly, however, it is also essential for the proper functioning of the scan detection system. Misidentifying legitimate traffic as malicious can lead to increased operational overhead and decreased trust in the system. Moreover, network administrators might receive a lot of false alerts, which can divert their attention from real threats.

Furthermore, it is important to consider the fact that web browsing tasks are usually the ones that involve links to **CDNs (Content Delivery Networks) and cloud services**. These platforms are engineered not only to maximize delivery efficiency through a globally distributed server network but, indeed, also to ensure users access the data from a server nearest to their location. Moreover, an essential feature of CDNs and particular cloud services is their tendency to misappropriate common DNS queries through their DNS-independent channels. CDNs are constructed on the principle of caching contents on several servers around the world. In a situation where a user requests data (e.g., a webpage, video, or image), first the request is routed to the nearest CDN server, which is usually faster than the original host. This routing often relies on IP addresses embedded in the content or HTTP headers, bypassing DNS resolution.

As an example, Direct Server Return (DSR) services make it possible for servers to get in touch with clients directly and use an IP address without any extra DNS lookups. This technique may

create problems for DNS-based detection systems, as it seems that clients are making connections with various IPs without first performing any DNS requests.

For a scan detection system to function effectively within an environment that heavily relies on CDNs and cloud services, it is critical to implement robust allowlisting mechanisms. These mechanisms should be dynamic, continuously learning from network traffic patterns to differentiate between legitimate DNS-independent connections and potential threats. By doing so, the system can focus on detecting real security risks while minimizing disruptions to normal network operations.

Chapter 7

Conclusion

This thesis has explored the development of a robust DNS-based scan detection system aimed at mitigating network security threats through real-time monitoring and defense mechanisms. By leveraging DNS traffic analysis and modifying TTL values, the system effectively reduces the impact of network scans, enhancing overall security posture. The research conducted has demonstrated the system's ability to disrupt unauthorized scanning activities, as evidenced by the significant delays imposed on network scans and the eventual blocking of probing attempts. Throughout this work, several key insights have emerged. The performance analysis highlights that the scan detection system operates efficiently with minimal resource consumption, making it suitable for integration into various network environments. The use of a dynamic DNS expiration table further contributes to the system's effectiveness, allowing it to maintain high performance even when handling large volumes of DNS traffic.

However, the research also identified certain limitations that warrant future improvements. The increased load on authoritative DNS servers due to smaller TTL values, the challenge of accurately allowlisting legitimate IP addresses, and the potential issues related to CDNs and cloud services are areas that require further refinement. Addressing these challenges will be critical in enhancing the system's accuracy and reducing false positives, particularly in complex network environments where DNS-independent connections are prevalent.

While the DNS-based detection system proves to be a valuable tool for identifying and mitigating network scanning threats, it is crucial to recognize that no single detection mechanism can provide comprehensive security on its own. The system would benefit from being integrated with other detection mechanisms and algorithms, such as signature-based detection, behavioral analysis, and machine learning-based anomaly detection. Combining these approaches would create a multi-layered defense strategy capable of addressing a broader range of threats, including those that may bypass DNS-based detection.

Looking forward, future work could focus on refining the system's allowlisting mechanisms, incorporating machine learning techniques to improve the identification of legitimate traffic. Additionally, expanding the system's capabilities to handle more sophisticated evasion techniques and adapting it to newer network architectures, such as those found in cloud-native environments, will be essential for maintaining its relevance in an evolving threat landscape.

In conclusion, this thesis has laid a strong foundation for the development of an advanced

scan detection system that balances security with performance. While challenges remain, the progress made thus far demonstrates the system's potential to significantly enhance network security and protect against emerging threats. Continued research and development in this area will be important in ensuring that the system remains effective and adaptable in the face of future challenges.

Appendix A: SDS Python Code

The following code implements the Scan Detection System (SDS) used for network scan detection in our experimental setup.

```
import netfilterqueue
import scapy.all as scapy
from scapy.layers import dns
from scapy.layers.inet6 import IPv6
import time
import pickle
import os
import psutil
import socket
import subprocess
import threading
import queue as queue_module

# hash table to save source ip, dns response's provided ip and new TTL
# associated

dns_expiration_table = {}

allowlisted_users = [
    '8.8.8.8', # Google DNS
    '8.8.4.4', # Google DNS
    '2001:4860:4860::8888', # Google DNS IPv6
    '2001:4860:4860::8844', # Google DNS IPv6
    '90.147.160.73',
    '45.135.106.143'
] # Example allowlisted users

allowlisted_domains = [
    '91.189.91.49', '91.189.91.96',
    '185.125.190.17', '185.125.190.49',
    '185.125.190.97', '91.189.91.98',
    '185.125.190.18', '185.125.190.98',
    '91.189.91.97', '185.125.190.48',
    '185.125.190.96', '91.189.91.48',
    '90.147.160.73', '45.135.106.143'
] # Example allowlisted domains
```

```

allowlisted_protocols = ['ICMP', 'SSH'] # Example allowlisted protocols

blocked_ips = {} # Dictionary to store blocked IP addresses and their
                  expiration times
block_duration = 1000 # Duration in seconds for which a machine should be
                      blocked

suspicious_count = {} # Global dictionary to track the number of suspicious
                      connections per IP
block_threshold = 100 # Set the threshold for blocking an IP after 100
                      suspicious connections

# Global variables for performance metrics
latencies = []
cpu_usages = []
memory_usages = []
packet_count = 0
start_time = time.time()

# File to store the DNS expiration table
dns_expiration_file = "dns_expiration_table.pkl"

def set_iptables_rule():
    try:
        subprocess.run(['iptables', '-I', 'FORWARD', '-j', 'NFQUEUE',
                        '--queue-num', '0'], check=True)
        print("iptables rule added successfully.")
    except subprocess.CalledProcessError as e:
        print(f"Failed to add iptables rule: {e}")

def empty_dns_expiration_table():
    with open('dns_expiration_table.pkl', 'wb') as f:
        pickle.dump({}, f)

```

```

def save_dns_expiration_table():
    with open(dns_expiration_file, 'wb') as f:
        pickle.dump(dns_expiration_table, f)

def load_dns_expiration_table():
    global dns_expiration_table
    if os.path.exists(dns_expiration_file):
        with open(dns_expiration_file, 'rb') as f:
            dns_expiration_table = pickle.load(f)

def get_source_ip(packet):
    if scapy.IPv6 in packet:
        return packet[scapy.IPv6].src
    elif scapy.IP in packet:
        return packet[scapy.IP].src
    else:
        return None

def get_destination_ip(packet):
    if scapy.IPv6 in packet:
        return packet[scapy.IPv6].dst
    elif scapy.IP in packet:
        return packet[scapy.IP].dst
    else:
        return None

def get_rdata_records(packet):
    rdata_records = []
    if packet.haslayer(scapy.DNS) and packet[scapy.DNS].qr == 1:
        for i in range(packet[scapy.DNS].ancount):
            rr = packet[scapy.DNS].an[i]

```

```

        if rr.type in [1, 28]: # 1 is A (IPv4), 28 is AAAA (IPv6)
            rdata_records.append(rr.rdata)
    return rdata_records

def modify_dns_response(packet):
    global dns_expiration_table
    current_time = time.time()
    tau = 300
    src_ip = get_destination_ip(packet)
    for i in range(packet[scapy.DNS].ancount):
        if packet[scapy.DNS].an[i].type in [1, 28]: # 1 is A (IPv4), 28 is AAAA
                                                    (IPv6)

            rdata = packet[scapy.DNS].an[i].rdata
            key = f"{src_ip}||{rdata}"
            dns_expiration_table[key] = current_time + tau
            packet[scapy.DNS].an[i].ttl = tau
    if scapy.IP in packet:
        del packet[scapy.IP].len
        del packet[scapy.IP].chksum
    elif scapy.IPv6 in packet:
        del packet[scapy.IPv6].plen
        del packet[scapy.IPv6].chksum
    if scapy.UDP in packet:
        del packet[scapy.UDP].len
        del packet[scapy.UDP].chksum
    return packet

def is_reverse_dns_query(packet):
    if packet.haslayer(scapy.DNSQR):
        for i in range(packet[scapy.DNS].qdcount):
            if packet[scapy.DNS].qd[i].qtype == 12: # 12 is the type code for
                                                    PTR records

                return True
    return False

def is_allowlisted_reverse_dns(packet):

```

```

src_ip = get_source_ip(packet)
if src_ip in allowlisted_users:
    return True
if packet.haslayer(scapy.DNSQR):
    for i in range(packet[scapy.DNS].qdcount):
        qname = packet[scapy.DNS].qd[i].qname.decode()
        for domain in allowlisted_domains:
            if domain in qname:
                return True
return False

def check_reverse_dns_query(packet):
    # Function to check if IP from reverse DNS query is in the hash table
    query_name = packet[scapy.DNS].qd[0].qname.decode()

    if query_name.endswith('.in-addr.arpa.'):
        reversed_ip = query_name[:-14].split('.')[::-1] # Adjusted to remove
                                                         the extra period

        # If it's an IP address in the reverse DNS query
        if all(part.isdigit() for part in reversed_ip):
            actual_ip = '.'.join(reversed_ip)
        else:
            # If it's a domain name in the reverse DNS query
            actual_ip = '.'.join(reversed_ip)
            try:
                resolved_ip = socket.gethostbyname(actual_ip)
                actual_ip = resolved_ip
            except socket.gaierror:
                return False

    src_ip = get_source_ip(packet)
    key = f"{src_ip}||{actual_ip}"

    # Check if the key is in the DNS expiration table
    if key in dns_expiration_table:
        current_time = time.time() # Get the current time
        expiration_time = dns_expiration_table[key] # Get the stored
                                                    expiration time

```

```

        # Compare current time with expiration time
        if current_time <= expiration_time:
            print("=====\n")
            print(f"Reverse-DNS query accepted since a connection
            from {src_ip} to {actual_ip} was previously established
            (so it is considered as legit)\n")
            print("=====\n")
            return True
        else:
            # The entry has expired, so it's no longer valid
            print("=====\n")
            print(f"Reverse-DNS query rejected since the
            connection from {src_ip} to {actual_ip} has expired\n")
            print("=====\n")
            return False

    return False

def calculate_metrics(start_time):

    # Measure performance
    end = time.time()
    latency = end - start_time
    latencies.append(latency)

    # Record CPU and memory usage
    cpu_usages.append(psutil.cpu_percent())
    memory_usages.append(psutil.virtual_memory().percent)

    # Print metrics periodically (every 10000 packets)
    if packet_count % 10000 == 0:
        avg_latency = sum(latencies) / len(latencies)
        avg_cpu = sum(cpu_usages) / len(cpu_usages)
        avg_memory = sum(memory_usages) / len(memory_usages)
        print("=====\n")
        print(f"Processed {packet_count} packets.")
        print(f"Average latency per packet: {avg_latency:.6f} seconds.")
        print(f"Average CPU usage: {avg_cpu:.2f}%")
        print(f"Average memory usage: {avg_memory:.2f}%")
        print("=====\n")

```

```

# Clear the metrics after printing to start fresh for the next 10000
                                packets

latencies.clear()
cpu_usages.clear()
memory_usages.clear()

def process_commands():
    while True:
        command = input("\nEnter a command: ")
        if command == "print_hash_table":
            print("\nDNS Expiration Table:")
            for key, value in dns_expiration_table.items():
                print(f"{key}: {time.strftime('%Y-%m-%d %H:%M:%S',
                    time.localtime(value))}")

        elif command == "block_ip":
            ip = input("Enter IP to block: ")
            blocked_ips[ip] = time.time() + block_duration
            print(f"Blocked IP: {ip}")

        elif command == "unblock_ip":
            ip = input("Enter IP to unblock: ")
            if ip in blocked_ips:
                del blocked_ips[ip]
                print(f"Unblocked IP: {ip}")
            else:
                print(f"IP {ip} not found in blocked list.")

        elif command == "add_allowlisted_user":
            ip = input("Enter IP to add to allowlist: ")
            allowlisted_users.append(ip)
            print(f"Added allowlisted user with IP: {ip}")

        elif command == "remove_allowlisted_user":
            ip = input("Enter IP to remove from allowlist: ")
            if ip in allowlisted_users:
                allowlisted_users.remove(ip)
                print(f"Removed allowlisted user with IP: {ip}")
            else:

```

```

        print(f"User with IP {ip} not found in allowlist.")

elif command == "add_allowlisted_domain":
    domain = input("Enter domain to add to allowlist: ")
    allowlisted_domains.append(domain)
    print(f"Added allowlisted domain: {domain}")

elif command == "remove_allowlisted_domain":
    domain = input("Enter domain to remove from allowlist: ")
    if domain in allowlisted_domains:
        allowlisted_domains.remove(domain)
        print(f"Removed allowlisted domain: {domain}")
    else:
        print(f"Domain {domain} not found in allowlist.")

elif command == "print_blocked_ips":
    print("\nBlocked IPs:")
    for ip, exp_time in blocked_ips.items():
        print(f"{ip}: {time.strftime('%Y-%m-%d %H:%M:%S',
            time.localtime(exp_time))}")

elif command == "print_allowlisted_users":
    print("\nAllowlisted users:")
    for ip in allowlisted_users:
        print(ip)

elif command == "print_allowlisted_domains":
    print("\nAllowlisted Domains:")
    for domain in allowlisted_domains:
        print(domain)

elif command == "exit":
    print("Exiting command interface.")
    break

elif command == "help":
    show_help()

else:
    print("Unknown command. Please try again.")

```



```

def show_help():
    # Display a list of all available commands and their descriptions
    print("\nAvailable commands:")
    print("1. print_hash_table: Display the current DNS expiration table.")
    print("2. block_ip: Block an IP address for a specified duration.")
    print("3. unblock_ip: Unblock an IP address.")
    print("4. add_allowlisted_user: Add a user to the allowlist.")
    print("5. remove_allowlisted_user: Remove a user from the allowlist.")
    print("6. add_allowlisted_domain: Add a domain to the allowlist.")
    print("7. remove_allowlisted_domain: Remove a domain from the allowlist.")
    print("8. print_blocked_ips: Display the list of currently blocked IPs.")
    print("9. print_allowlisted_users: Display the list of currently allowlisted
            users.")
    print("10. print_allowlisted_domains: Display the list of currently
            allowlisted domains.")
    print("11. help: Display this help message with all available commands.")
    print("12. exit: Exit the command interface.")

def process_packet(packet):

    global latencies, cpu_usages, memory_usages, packet_count, blocked_ips
    packet_count += 1
    # Measure latency
    start = time.time()
    scapy_packet = scapy.IP(packet.get_payload())
    current_time = time.time()

    # Remove expired blocked entries
    blocked_ips = {ip: exp_time for ip, exp_time in blocked_ips.items()
                    if exp_time > current_time}

    if scapy.IPv6 in scapy_packet:
        scapy_packet = scapy.IPv6(packet.get_payload())

    # Flag to indicate if a verdict has been given
    verdict_given = False

```

```

if scapy_packet.haslayer(scapy.DNSRR) and
(scapy.IP in scapy_packet or scapy.IPv6 in scapy_packet):
    # Process DNS response
    rdata_records = get_rdata_records(scapy_packet)
    #check if there are any rdata records
    if rdata_records:
        modified_answers = modify_dns_response(scapy_packet)
        packet.set_payload(bytes(modified_answers))
        packet.accept()
        verdict_given = True

elif is_reverse_dns_query(scapy_packet):

    if check_reverse_dns_query(scapy_packet) or
    is_allowlisted_reverse_dns(scapy_packet):
        # accept reverse dns queries only if allowed
        packet.accept()
        print("\n=====\\n")
        print("Reverse-DNS query accepted")
        print("\n=====\\n")
    else:
        src_ip = get_source_ip(scapy_packet)
        # log and drop not expected reverse dns queries
        print("\n
=====\\n")
        print(f"Reverse-DNS query from {src_ip} detected.
Blocking this dns reverse query.")
        print("\n
=====\\n")
        packet.drop()

    verdict_given = True

elif scapy_packet.haslayer(scapy.DNSQR):
    # Accept all DNS query packets
    packet.accept()
    verdict_given = True

if not verdict_given and (scapy.IP in scapy_packet or scapy.IPv6 in
scapy_packet):

    # Process communication attempt

```

```

src_ip = get_source_ip(scapy_packet)
dst_ip = get_destination_ip(scapy_packet)
key = f"{src_ip}||{dst_ip}"
key2 = f"{dst_ip}||{src_ip}"

if src_ip in blocked_ips:
    print("\n
    *****\n")
    print(f"Connection attempt from blocked IP {src_ip}
    to {dst_ip} blocked.")
    print("\n
    *****\n")
    packet.drop()
    verdict_given = True

if not verdict_given and (key in dns_expiration_table
and dns_expiration_table[key] > current_time) or
(key2 in dns_expiration_table and dns_expiration_table[key2] >
current_time):
    packet.accept() # Allow the connection
    verdict_given = True

# Allowlist specific protocols and applications
if not verdict_given:
    if scapy_packet.haslayer(scapy.ICMP) or
    scapy_packet.haslayer(scapy.TCP) and
    (scapy_packet[scapy.TCP].dport == 22 or
    scapy_packet[scapy.TCP].sport == 22):
        # ICMP or SSH (port 22)
        packet.accept()
        verdict_given = True

if not verdict_given:

    # Initialize the count for the src_ip if it's not already in the
    dictionary

    if src_ip not in suspicious_count:
        suspicious_count[src_ip] = 0

```

```
# Check if the connection involves allowlisted IPs
if src_ip in allowlisted_users or
dst_ip in allowlisted_users:
    packet.accept()
    # Allowlisted users are not considered malicious
    verdict_given = True

else:
    # Increment the suspicious count for the source IP
    suspicious_count[src_ip] += 1

    if suspicious_count[src_ip] >= block_threshold:
        # Block the IP if it exceeds the threshold
        print("\n\n
+-+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--\n\n")
        print(f"Blocking IP {src_ip} after
{suspicious_count[src_ip]} suspicious connections.")
        print("\n\n
+-+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--\n\n")
        blocked_ips[src_ip] = current_time + block_duration
        packet.drop() # Drop the connection
        verdict_given = True
    else:
        # Log the suspicious connection but do not block it (just
                                log it)

        print("\n
*****\n")
        print(f"Suspicious connection from {src_ip}
to {dst_ip} detected.
Count: {suspicious_count[src_ip]}")
        print("\n
*****\n")
        packet.accept()
        verdict_given = True

# Call function to calculate metrics
calculate_metrics(start)

queue = netfilterqueue.NetfilterQueue()
# loading the locally saved hash table
load_dns_expiration_table()
```

```

# run iptables rules on the startup of the script
set_iptables_rule()
# Clear the file at the start (uncomment the next line if you want to clear the
                                file)

#empty_dns_expiration_table()
queue.bind(0, process_packet)


# Define a thread-safe queue for inter-thread communication
command_queue = queue_module.Queue()
# Start the command processing thread
command_thread = threading.Thread(target=process_commands)
command_thread.daemon = True
command_thread.start()


try:
    queue.run()

except KeyboardInterrupt:
    # saving hash table before exiting
    save_dns_expiration_table()
    print("DNS expiration table saved.")

```

Appendix B: Nmap Scan Results

This appendix contains the full output from the Nmap scan performed by VM2 during the experiment.

```
Starting Nmap 7.94SVN ( https://nmap.org ) at 2024-08-15 16:00 CEST
NSE: Loaded 156 scripts for scanning.
NSE: Script Pre-scanning.
Initiating NSE at 16:00
Completed NSE at 16:00, 0.00s elapsed
Initiating NSE at 16:00
Completed NSE at 16:00, 0.00s elapsed
Initiating NSE at 16:00
Completed NSE at 16:00, 0.00s elapsed
Initiating Ping Scan at 16:00
Scanning 17 hosts [4 ports/host]
Completed Ping Scan at 16:00, 1.28s elapsed (17 total hosts)
Initiating Parallel DNS resolution of 17 hosts. at 16:00
Completed Parallel DNS resolution of 17 hosts. at 16:00, 13.00s elapsed
Initiating SYN Stealth Scan at 16:00
Scanning 8 hosts [1024 ports/host]
Discovered open port 443/tcp on 54.82.22.214
SYN Stealth Scan Timing: About 3.09% done; ETC: 16:17 (0:16:13 remaining)
SYN Stealth Scan Timing: About 3.34% done; ETC: 16:31 (0:29:23 remaining)
SYN Stealth Scan Timing: About 3.60% done; ETC: 16:42 (0:40:36 remaining)
SYN Stealth Scan Timing: About 3.88% done; ETC: 16:52 (0:49:56 remaining)
SYN Stealth Scan Timing: About 4.16% done; ETC: 17:01 (0:58:02 remaining)
SYN Stealth Scan Timing: About 4.43% done; ETC: 17:08 (1:05:09 remaining)
SYN Stealth Scan Timing: About 4.69% done; ETC: 17:15 (1:11:24 remaining)
SYN Stealth Scan Timing: About 4.97% done; ETC: 17:21 (1:16:50 remaining)
Increasing send delay for 52.16.210.4 from 0 to 5 due to 11 out of 11 dropped
probes since last increase.
SYN Stealth Scan Timing: About 5.23% done; ETC: 17:27 (1:21:50 remaining)
SYN Stealth Scan Timing: About 5.50% done; ETC: 17:31 (1:26:12 remaining)
Increasing send delay for 54.82.22.214 from 0 to 5 due to 11 out of 13 dropped
probes since last increase.
SYN Stealth Scan Timing: About 5.82% done; ETC: 17:37 (1:30:51 remaining)
SYN Stealth Scan Timing: About 6.21% done; ETC: 17:43 (1:36:06 remaining)
SYN Stealth Scan Timing: About 6.73% done; ETC: 17:49 (1:41:22 remaining)
SYN Stealth Scan Timing: About 7.39% done; ETC: 17:56 (1:46:48 remaining)
Increasing send delay for 52.16.210.4 from 5 to 10 due to 11 out of 11 dropped
probes since last increase.
SYN Stealth Scan Timing: About 8.24% done; ETC: 18:03 (1:52:40 remaining)
```

```

Increasing send delay for 54.82.22.214 from 5 to 10 due to 11 out of 11 dropped
                                probes since last increase.
SYN Stealth Scan Timing: About 9.56% done; ETC: 18:12 (1:58:55 remaining)
SYN Stealth Scan Timing: About 11.97% done; ETC: 18:23 (2:05:31 remaining)
SYN Stealth Scan Timing: About 29.72% done; ETC: 18:49 (1:58:22 remaining)
SYN Stealth Scan Timing: About 35.86% done; ETC: 18:52 (1:49:55 remaining)
SYN Stealth Scan Timing: About 41.55% done; ETC: 18:54 (1:41:19 remaining)
SYN Stealth Scan Timing: About 47.05% done; ETC: 18:55 (1:32:36 remaining)
Completed SYN Stealth Scan against 44.228.249.3 in 5347.22s (7 hosts left)
SYN Stealth Scan Timing: About 52.42% done; ETC: 18:56 (1:23:51 remaining)
SYN Stealth Scan Timing: About 57.79% done; ETC: 18:58 (1:15:00 remaining)
SYN Stealth Scan Timing: About 63.06% done; ETC: 18:59 (1:06:06 remaining)
SYN Stealth Scan Timing: About 68.24% done; ETC: 19:00 (0:57:09 remaining)
Completed SYN Stealth Scan against 188.114.97.7 in 7658.62s (6 hosts left)
Completed SYN Stealth Scan against 52.16.210.4 in 7702.97s (5 hosts left)
Completed SYN Stealth Scan against 44.238.29.244 in 7786.10s (4 hosts left)
SYN Stealth Scan Timing: About 73.05% done; ETC: 18:59 (0:48:06 remaining)
Increasing send delay for 34.249.203.140 from 0 to 5 due to 11 out of 11 dropped
                                probes since last increase.
Increasing send delay for 34.249.203.140 from 5 to 10 due to 11 out of 11
                                dropped probes since last increase.
SYN Stealth Scan Timing: About 77.61% done; ETC: 18:55 (0:39:07 remaining)
Completed SYN Stealth Scan against 54.82.22.214 in 8314.27s (3 hosts left)
SYN Stealth Scan Timing: About 82.24% done; ETC: 18:51 (0:30:16 remaining)
SYN Stealth Scan Timing: About 86.91% done; ETC: 18:46 (0:21:39 remaining)
Completed SYN Stealth Scan against 216.58.209.52 in 8637.30s (2 hosts left)
SYN Stealth Scan Timing: About 91.72% done; ETC: 18:41 (0:13:21 remaining)
SYN Stealth Scan Timing: About 96.65% done; ETC: 18:37 (0:05:16 remaining)
Completed SYN Stealth Scan against 216.58.204.148 in 9145.58s (1 host left)
Completed SYN Stealth Scan at 18:35, 9313.65s elapsed (8192 total ports)
Initiating Service scan at 18:35
Scanning 1 service on 8 hosts
Completed Service scan at 18:36, 5.01s elapsed (1 service on 8 hosts)
Initiating OS detection (try #1) against 8 hosts
Retrying OS detection (try #2) against 8 hosts
Initiating Traceroute at 18:36
Completed Traceroute at 18:37, 70.47s elapsed
Initiating Parallel DNS resolution of 1 host. at 18:37
Completed Parallel DNS resolution of 1 host. at 18:37, 13.00s elapsed
NSE: Script scanning 8 hosts.
Initiating NSE at 18:37
Completed NSE at 18:38, 60.10s elapsed

```

```

Initiating NSE at 18:38
Completed NSE at 18:40, 120.08s elapsed
Initiating NSE at 18:40
Completed NSE at 18:40, 0.01s elapsed
Nmap scan report for 44.228.249.3
Host is up (0.26s latency).
All 1024 scanned ports on 44.228.249.3 are in ignored states.
Not shown: 1024 filtered tcp ports (no-response)
Too many fingerprints match this host to give specific OS details

TRACEROUTE (using proto 1/icmp)
HOP RTT      ADDRESS
1    0.54 ms  192.168.100.10
2    ... 30

Nmap scan report for 44.238.29.244
Host is up (0.25s latency).
All 1024 scanned ports on 44.238.29.244 are in ignored states.
Not shown: 1024 filtered tcp ports (no-response)
Too many fingerprints match this host to give specific OS details

TRACEROUTE (using proto 1/icmp)
HOP RTT      ADDRESS
-    Hop 1 is the same as for 44.228.249.3
2    ... 30

Nmap scan report for 52.16.210.4
Host is up (0.082s latency).
All 1024 scanned ports on 52.16.210.4 are in ignored states.
Not shown: 1024 filtered tcp ports (no-response)
Too many fingerprints match this host to give specific OS details

TRACEROUTE (using port 443/tcp)
HOP RTT      ADDRESS
-    Hop 1 is the same as for 44.228.249.3
2    ... 30

Nmap scan report for 216.58.209.52
Host is up (0.030s latency).
All 1024 scanned ports on 216.58.209.52 are in ignored states.
Not shown: 1024 filtered tcp ports (no-response)
Too many fingerprints match this host to give specific OS details

```



```

TRACEROUTE (using proto 1/icmp)
HOP RTT      ADDRESS
-   Hop 1 is the same as for 44.228.249.3
2   ... 30

Nmap scan report for 34.249.203.140
Host is up (0.066s latency).
All 1024 scanned ports on 34.249.203.140 are in ignored states.
Not shown: 1024 filtered tcp ports (no-response)
Too many fingerprints match this host to give specific OS details

TRACEROUTE (using port 443/tcp)
HOP RTT      ADDRESS
-   Hop 1 is the same as for 44.228.249.3
2   ... 30

Nmap scan report for 216.58.204.148
Host is up (0.031s latency).
All 1024 scanned ports on 216.58.204.148 are in ignored states.
Not shown: 1024 filtered tcp ports (no-response)
Too many fingerprints match this host to give specific OS details

TRACEROUTE (using proto 1/icmp)
HOP RTT      ADDRESS
-   Hop 1 is the same as for 44.228.249.3
2   ... 30

Nmap scan report for 188.114.97.7
Host is up (0.032s latency).
All 1024 scanned ports on 188.114.97.7 are in ignored states.
Not shown: 1024 filtered tcp ports (no-response)
Too many fingerprints match this host to give specific OS details

TRACEROUTE (using proto 1/icmp)
HOP RTT      ADDRESS
-   Hop 1 is the same as for 44.228.249.3
2   ... 30

Nmap scan report for 54.82.22.214
Host is up (0.17s latency).
Not shown: 1023 filtered tcp ports (no-response)

```

```

PORT      STATE SERVICE      VERSION
443/tcp   open  tcpwrapped

Warning: OSScan results may be unreliable because we could not find at least 1
open and 1 closed port

OS fingerprint not ideal because: Missing a closed TCP port so results
incomplete

No OS matches for host

TRACEROUTE (using port 443/tcp)
HOP RTT      ADDRESS
-   Hop 1 is the same as for 44.228.249.3
2   ... 30

Initiating SYN Stealth Scan at 18:40
Scanning 9 hosts [1024 ports/host]
SYN Stealth Scan Timing: About 0.74% done
SYN Stealth Scan Timing: About 1.00% done; ETC: 20:22 (1:40:17 remaining)
SYN Stealth Scan Timing: About 1.27% done; ETC: 20:39 (1:57:26 remaining)
SYN Stealth Scan Timing: About 1.55% done; ETC: 20:51 (2:08:25 remaining)
SYN Stealth Scan Timing: About 1.86% done; ETC: 20:59 (2:15:46 remaining)
SYN Stealth Scan Timing: About 2.23% done; ETC: 21:07 (2:23:14 remaining)
SYN Stealth Scan Timing: About 3.01% done; ETC: 21:16 (2:30:36 remaining)
Increasing send delay for 54.204.37.212 from 0 to 5 due to 11 out of 11 dropped
probes since last increase.
Increasing send delay for 178.79.134.182 from 0 to 5 due to 11 out of 11 dropped
probes since last increase.
Increasing send delay for 54.73.53.134 from 0 to 5 due to 11 out of 11 dropped
probes since last increase.
Increasing send delay for 52.1.112.43 from 0 to 5 due to 11 out of 11 dropped
probes since last increase.
Increasing send delay for 107.20.213.223 from 0 to 5 due to 11 out of 11 dropped
probes since last increase.
SYN Stealth Scan Timing: About 5.50% done; ETC: 21:28 (2:38:29 remaining)
Increasing send delay for 54.204.37.212 from 5 to 10 due to 11 out of 11 dropped
probes since last increase.
Increasing send delay for 178.79.134.182 from 5 to 10 due to 11 out of 11
dropped probes since last increase.
Increasing send delay for 54.73.53.134 from 5 to 10 due to 11 out of 11 dropped
probes since last increase.
Increasing send delay for 52.1.112.43 from 5 to 10 due to 11 out of 11 dropped
probes since last increase.
Increasing send delay for 107.20.213.223 from 5 to 10 due to 11 out of 11

```

```

dropped probes since last increase.
SYN Stealth Scan Timing: About 15.16% done; ETC: 21:37 (2:30:06 remaining)
SYN Stealth Scan Timing: About 20.79% done; ETC: 21:39 (2:21:13 remaining)
SYN Stealth Scan Timing: About 26.22% done; ETC: 21:40 (2:12:11 remaining)
SYN Stealth Scan Timing: About 31.43% done; ETC: 21:40 (2:03:10 remaining)
SYN Stealth Scan Timing: About 36.59% done; ETC: 21:40 (1:54:09 remaining)
SYN Stealth Scan Timing: About 41.72% done; ETC: 21:41 (1:45:06 remaining)
SYN Stealth Scan Timing: About 46.78% done; ETC: 21:41 (1:36:05 remaining)
SYN Stealth Scan Timing: About 51.86% done; ETC: 21:41 (1:26:59 remaining)
SYN Stealth Scan Timing: About 56.91% done; ETC: 21:41 (1:17:55 remaining)
SYN Stealth Scan Timing: About 61.95% done; ETC: 21:41 (1:08:51 remaining)
SYN Stealth Scan Timing: About 66.99% done; ETC: 21:41 (0:59:46 remaining)
SYN Stealth Scan Timing: About 72.03% done; ETC: 21:42 (0:50:40 remaining)
SYN Stealth Scan Timing: About 77.07% done; ETC: 21:42 (0:41:34 remaining)
SYN Stealth Scan Timing: About 82.10% done; ETC: 21:42 (0:32:27 remaining)
SYN Stealth Scan Timing: About 87.11% done; ETC: 21:42 (0:23:23 remaining)
SYN Stealth Scan Timing: About 92.13% done; ETC: 21:42 (0:14:17 remaining)
Completed SYN Stealth Scan against 65.61.137.117 in 10392.96s (8 hosts left)
Completed SYN Stealth Scan against 54.204.37.212 in 10401.60s (7 hosts left)
Completed SYN Stealth Scan against 52.1.112.43 in 10406.59s (6 hosts left)
Completed SYN Stealth Scan against 69.164.223.208 in 10408.97s (5 hosts left)
Completed SYN Stealth Scan against 178.79.134.182 in 10409.90s (4 hosts left)
Completed SYN Stealth Scan against 54.73.53.134 in 10415.57s (3 hosts left)
Completed SYN Stealth Scan against 107.20.213.223 in 10422.58s (2 hosts left)
SYN Stealth Scan Timing: About 97.14% done; ETC: 21:41 (0:05:10 remaining)
Completed SYN Stealth Scan against 192.124.249.5 in 10785.71s (1 host left)
Completed SYN Stealth Scan at 21:40, 10789.02s elapsed (9216 total ports)
Initiating Service scan at 21:40
Initiating OS detection (try #1) against 9 hosts
Retrying OS detection (try #2) against 9 hosts
Initiating Traceroute at 21:41
Completed Traceroute at 21:42, 79.44s elapsed
Initiating Parallel DNS resolution of 1 host. at 21:42
Completed Parallel DNS resolution of 1 host. at 21:42, 13.00s elapsed
NSE: Script scanning 9 hosts.
Initiating NSE at 21:42
Completed NSE at 21:42, 5.09s elapsed
Initiating NSE at 21:42
Completed NSE at 21:42, 0.01s elapsed
Initiating NSE at 21:42
Completed NSE at 21:42, 0.01s elapsed
Nmap scan report for 65.61.137.117

```

Host **is** up (0.16s latency).
All 1024 scanned ports on 65.61.137.117 are **in** ignored states.
Not shown: 1024 filtered tcp ports (no-response)
Too many fingerprints match this host to give specific OS details

TRACEROUTE (using proto 1/icmp)

HOP RTT ADDRESS

- Hop 1 **is** the same as **for** 44.228.249.3
2 ... 30

Nmap scan report **for** 54.73.53.134

Host **is** up (0.074s latency).
All 1024 scanned ports on 54.73.53.134 are **in** ignored states.
Not shown: 1024 filtered tcp ports (no-response)
Too many fingerprints match this host to give specific OS details

TRACEROUTE (using port 80/tcp)

HOP RTT ADDRESS

- Hop 1 **is** the same as **for** 44.228.249.3
2 ... 30

Nmap scan report **for** 69.164.223.171

Host **is** up (0.13s latency).
All 1024 scanned ports on 69.164.223.171 are **in** ignored states.
Not shown: 1024 filtered tcp ports (no-response)
Too many fingerprints match this host to give specific OS details

TRACEROUTE (using proto 1/icmp)

HOP RTT ADDRESS

- Hop 1 **is** the same as **for** 44.228.249.3
2 ... 30

Nmap scan report **for** 69.164.223.208

Host **is** up (0.14s latency).
All 1024 scanned ports on 69.164.223.208 are **in** ignored states.
Not shown: 1024 filtered tcp ports (no-response)
Too many fingerprints match this host to give specific OS details

TRACEROUTE (using proto 1/icmp)

HOP RTT ADDRESS

- Hop 1 **is** the same as **for** 44.228.249.3
2 ... 30

Nmap scan report for 54.204.37.212
Host is up (0.14s latency).
All 1024 scanned ports on 54.204.37.212 are in ignored states.
Not shown: 1024 filtered tcp ports (no-response)
Too many fingerprints match this host to give specific OS details

TRACEROUTE (using port 443/tcp)
HOP RTT ADDRESS
- Hop 1 is the same as for 44.228.249.3
2 ... 30

Nmap scan report for 107.20.213.223
Host is up (0.14s latency).
All 1024 scanned ports on 107.20.213.223 are in ignored states.
Not shown: 1024 filtered tcp ports (no-response)
Too many fingerprints match this host to give specific OS details

TRACEROUTE (using port 443/tcp)
HOP RTT ADDRESS
- Hop 1 is the same as for 44.228.249.3
2 ... 30

Nmap scan report for 52.1.112.43
Host is up (0.13s latency).
All 1024 scanned ports on 52.1.112.43 are in ignored states.
Not shown: 1024 filtered tcp ports (no-response)
Too many fingerprints match this host to give specific OS details

TRACEROUTE (using port 80/tcp)
HOP RTT ADDRESS
- Hop 1 is the same as for 44.228.249.3
2 ... 30

Nmap scan report for 192.124.249.5
Host is up (0.034s latency).
All 1024 scanned ports on 192.124.249.5 are in ignored states.
Not shown: 1024 filtered tcp ports (no-response)
Too many fingerprints match this host to give specific OS details

TRACEROUTE (using proto 1/icmp)
HOP RTT ADDRESS

```
- Hop 1 is the same as for 44.228.249.3
2 ... 30
```

Nmap scan report for 178.79.134.182

Host is up (0.055s latency).

All 1024 scanned ports on 178.79.134.182 are in ignored states.

Not shown: 1024 filtered tcp ports (no-response)

Too many fingerprints match this host to give specific OS details

TRACEROUTE (using port 443/tcp)

HOP RTT ADDRESS

```
- Hop 1 is the same as for 44.228.249.3
2 ... 30
```

NSE: Script Post-scanning.

Initiating NSE at 21:42

Completed NSE at 21:42, 0.00s elapsed

Initiating NSE at 21:42

Completed NSE at 21:42, 0.00s elapsed

Initiating NSE at 21:42

Completed NSE at 21:42, 0.00s elapsed

Read data files from: /usr/bin/../share/nmap

OS and Service detection performed. Please report any incorrect results at <https://nmap.org/submit/> .

Nmap done: 17 IP addresses (17 hosts up) scanned in 20541.19 seconds

Raw packets sent: 39196 (1.788MB) | Rcvd: 48 (2.260KB)

Bibliography

1. Cricket Liu, Paul Albitz, DNS and BIND *5th Edition*
2. P.V. Mockapetris. Domain names: Concepts and facilities. *RFC 882, November 1983. Obsoleted by RFCs 1034, 1035, updated by RFC 973*
3. P.V. Mockapetris. Domain names: Implementation specification. *RFC 883, November 1983. Obsoleted by RFCs 1034, 1035, updated by RFC 973*
4. P.V. Mockapetris. Domain names - concepts and facilities. *RFC 1034 (INTERNET STANDARD), November 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 2065, 2181, 2308, 2535, 4033, 4034, 4035, 4343, 4035, 4592, 5936*
5. P.V. Mockapetris. Domain names - implementation and specification. *RFC 1035 (INTERNET STANDARD), November 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2673, 2845, 3425, 3658, 4033, 4034, 4035, 4343, 5936, 5966, 6604*
6. W. Stallings, Network Security Essentials: Applications and Standards *6th ed. Pearson, 2017*
7. G. F. Lyon, Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning. *Nmap Project, 2009*
8. C. Leckie and R. Kotagiri, "A probabilistic approach to detecting network scans" *2002, pp. 359-372*
9. M. de Vivo, E. Carrasco, G. Isern, and G. O. de Vivo, "A review of port scanning techniques" *ACM SIGCOMM Computer Communication Review, vol. 29, pp. 41-48, 1999*
10. Ajay Singh Chauhan, Practical Network Scanning: Capture network vulnerability. *2018*
11. J. Erickson, Hacking: The Art of Exploitation, *2nd ed. San Francisco, CA: No Starch Press, 2008.*
12. C. McNab, Network Security Assessment: Know Your Network, *3rd ed. Sebastopol, CA: O'Reilly Media, 2016.*
13. R. Davis, The Art of Network Penetration Testing: Taking over any company in the world step by step. *Birmingham, UK: Packt Publishing, 2021.*
14. Open Web Application Security Project (OWASP). *OWASP. [Online]. [Available: <https://owasp.org/>]*
15. SANS Institute. *SANS Institute Information Security Reading Room.[Online]. [Available: <https://www.sans.org/>]*
16. Cisco Networking Academy. *Cisco Networking Academy Program. [Online]. [Available: <https://www.netacad.com/>]*

17. Wireshark. *Wireshark: Go Deep*. [Online]. [Available: <https://www.wireshark.org/>]
18. E. D. Zwicky, S. Cooper, and D. B. Chapman, Building Internet Firewalls, *2nd ed.* O'Reilly Media, 2000.

Citations

- [1] SuperScan - Wikipedia, The Free Encyclopedia [<https://en.wikipedia.org/wiki/SuperScan>]
- [2] Tenable Network Security. Nessus [<https://www.tenable.com/products/nessus>]
- [3] Greenbone Networks GmbH. OpenVAS [<https://www.openvas.org/>]
- [4] Robert David Graham. Masscan [<https://github.com/robertdavidgraham/masscan>]
- [5] Gordon Lyon. Zenmap [<https://nmap.org/zenmap/>]
- [6] G. F. Lyon, Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning [<https://nmap.org/>]
- [7] David Whyte, P.C. van Oorschot, and Evangelos Kranakis, "ARP-based Detection of Scanning Worms Within an Enterprise Network" [Online]. Available: [<https://people.scs.carleton.ca/~kranakis/Papers/TR-05-02.pdf>]
- [8] Yong-lin Zhou, Qing-shan Li, Qidi Miao, and Kangbin Yim."DGA-Based Botnet Detection Using DNS Traffic" [Online]. [Online]. Available: [<https://isyou.info/jisis/vol3/no34/jisis-2013-vol3-no34-11.pdf>]
- [9] K. Ilgun, R. A. Kemmerer and P. A. Porras, "State transition analysis: a rule-based intrusion detection approach," in IEEE Transactions on Software Engineering, vol. 21, no. 3, pp. 181-199, March 1995, doi: 10.1109/32.372146. [Online]. Available: [<https://ieeexplore.ieee.org/abstract/document/372146>]
- [10] Vigna, Giovanni and Kemmerer, Richard A. 'NetSTAT: A Network-based Intrusion Detection System'. 1 Jan. 1999 : 37 – 71. [Online]. Available: [<https://content.iospress.com/articles/journal-of-computer-security/jcs116>]
- [11] Rieck, K., Laskov, P. Language models for detection of unknown attacks in network traffic. J Comput Virol 2, 243–256 (2007). [Online]. Available: [<https://doi.org/10.1007/s11416-006-0030-0>]
- [12] U. Lindqvist and P. A. Porras, "Detecting computer and network misuse through the production-based expert system toolset (P-BEST)," Proceedings of the 1999 IEEE Symposium on Security and Privacy (Cat. No.99CB36344), Oakland, CA, USA, 1999, pp. 146-161, doi: 10.1109/SECPRI.1999.766911. [Online]. Available: [<https://ieeexplore.ieee.org/abstract/document/766911>]
- [13] M. Roesch, "Snort: Lightweight Intrusion Detection for Networks," [Online]. Available: [https://www.usenix.org/legacy/publications/library/proceedings/lisa99/full_papers/roesch/roesch.pdf]
- [14] OISF Development Team, "Suricata: Open Source Intrusion Detection and Prevention System," [Online]. Available: [<https://suricata.io>]

- [15] The Zeek Project, "Zeek: Open Source Network Security Monitor," [Online]. Available: [<https://zeek.org>]
- [16] Gerhard Munz, Sa Li, Georg Carle. Computer Networks and Internet. Wilhelm Schickard Institute for Computer Science. University of Tuebingen, Germany. "Traffic Anomaly Detection Using K-Means Clustering". [Online]. Available: [<https://www.net.in.tum.de/projects/dfg-lupus/files/muenz07k-means.pdf>]
- [17] Gilberto Fernandes, Luiz F. Carvalho, Joel J.P.C. Rodrigues, Mario Lemes Proença, "Network anomaly detection using IP flows with Principal Component Analysis and Ant Colony Optimization", Journal of Network and Computer Applications, Volume 64, 2016. [Online]. Available: [<https://www.sciencedirect.com/science/article/pii/S1084804516000618>]
- [18] Chen, Y., Zhang, J., Yeo, C.K. (2020). Network Anomaly Detection Using Federated Deep Autoencoding Gaussian Mixture Model. In: Boumerdassi, S., Renault, É., Mühlethaler, P. (eds) Machine Learning for Networking. MLN 2019. Lecture Notes in Computer Science(), vol 12081. Springer, Cham. [Online]. Available: [https://doi.org/10.1007/978-3-030-45778-5_1]
- [19] Carlos A. Catania, Facundo Bromberg, Carlos García Garino, "An autonomous labeling approach to support vector machines algorithms for network traffic anomaly detection", Expert Systems with Applications, Volume 39, Issue 2, 2012. [Online]. Available: [<https://www.sciencedirect.com/science/article/pii/S0957417411011808>]
- [20] Tao X, Peng Y, Zhao F, Zhao P, Wang Y. A parallel algorithm for network traffic anomaly detection based on Isolation Forest. International Journal of Distributed Sensor Networks. 2018;14(11). doi:10.1177/1550147718814471. [Online]. Available: [<https://journals.sagepub.com/doi/full/10.1177/1550147718814471>]
- [21] S. Naseer et al., "Enhanced Network Anomaly Detection Based on Deep Neural Networks," in IEEE Access, vol. 6, pp. 48231-48246, 2018, doi: 10.1109/ACCESS.2018.2863036. [Online]. Available: [<https://ieeexplore.ieee.org/abstract/document/8438865>]
- [22] Gao, D., Reiter, M.K., Song, D. (2006). Behavioral Distance Measurement Using Hidden Markov Models. In: Zamboni, D., Kruegel, C. (eds) Recent Advances in Intrusion Detection. RAID 2006. Lecture Notes in Computer Science, vol 4219. Springer, Berlin, Heidelberg. [Online]. Available: [https://doi.org/10.1007/11856214_2]
- [23] Bontemps, L., Cao, V.L., McDermott, J., Le-Khac, NA. (2016). Collective Anomaly Detection Based on Long Short-Term Memory Recurrent Neural Networks. In: Dang, T., Wagner, R., Küng, J., Thoai, N., Takizawa, M., Neuhold, E. (eds) Future Data and Security Engineering. FDSE 2016. Lecture Notes in Computer Science(), vol 10018. Springer, Cham. [Online]. Available: [https://doi.org/10.1007/978-3-319-48057-2_9]

- [24] W. Tylman, "Anomaly-Based Intrusion Detection Using Bayesian Networks," 2008 Third International Conference on Dependability of Computer Systems DepCoS-RELCOMEX, Szklarska Poreba, Poland, 2008, pp. 211-218, doi: 10.1109/DepCoS-RELCOMEX.2008.52. [Online]. Available: [<https://ieeexplore.ieee.org/abstract/document/4573059>]
- [25] J. Zhang, M. Zulkernine and A. Haque, "Random-Forests-Based Network Intrusion Detection Systems," in IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews), vol. 38, no. 5, pp. 649-659, Sept. 2008, doi: 10.1109/TSMCC.2008.923876. [Online]. Available: [<https://ieeexplore.ieee.org/abstract/document/4603103>]
- [26] "ARPANET" - Wikipedia, The Free Encyclopedia. [Online]. Available: [<https://en.wikipedia.org/wiki/ARPANET>]
- [27] ICANN (Internet Corporation for Assigned Names and Numbers). [Online]. Available: [<https://www.icann.org/>]
- [28] List of Root Servers. IANA (Internet Assigned Numbers Authority). [Online]. Available: [<https://www.iana.org/domains/root/servers>]
- [29] D. Whyte, E. Kranakis, and P. C. Van Oorschot, "DNS-based detection of scanning worms in an enterprise network" in *Proc. NDSS, 2005*, pp. 1-17.
- [30] M. A. Ahmad and S. Woodhead, "Containment of fast scanning computer network worms", in *Proc. Int. Conf. Internet Distrib. Comput. Syst. Cham, Switzerland: Springer, 2015*, pp. 235-247.
- [31] J. H. Jafarian, M. Abolfathi and M. Rahimian, "Detecting Network Scanning Through Monitoring and Manipulation of DNS Traffic," in *IEEE Access*, vol. 11, pp. 20267-20283, 2023, doi: 10.1109/ACCESS.2023.3250106. [Online]. Available: [<https://ieeexplore.ieee.org/document/10054395>]
- [32] Reconnaissance Techniques; Active Scanning: Scanning IP Blocks. [Online]. Available: [<https://attack.mitre.org/techniques/T1595/001/>]
- [33] Top-1000000-domains, Github. [Online]. Available: [<https://github.com/zer0h/top-1000000-domains/blob/master/top-10000-domains>]
- [34] Awesome Vulnerable Web Applications List. [Online]. Available: [<https://github.com/geeksonsecurity/vuln-web-apps?tab=readme-ov-file>]